RICE UNIVERSITY

Optics and Virtualization as Data Center Network Infrastructure

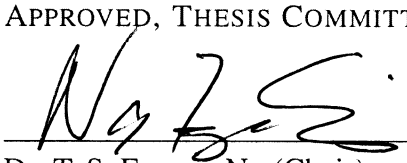
by

Guohui Wang

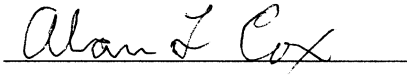
A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

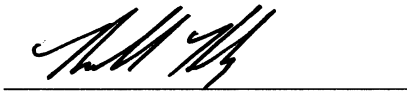


Dr. T. S. Eugene Ng (Chair),
Associate Professor,
Computer Science

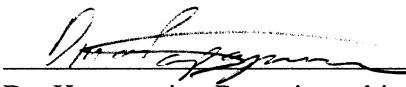


Dr. Alan L. Cox,
Associate Professor,
Computer Science

Dr. Edward W. Knightly,
Professor,
Electrical and Computer Engineering



Dr. Michael E. Kaminsky,
Senior Research Scientist,
Intel Labs



Dr. Konstantina Papagiannaki,
Principal Researcher,
Telefonica Research

HOUSTON, TEXAS
JULY, 2011

Optics and Virtualization as Data Center Network Infrastructure

Guohui Wang

Abstract

The emerging cloud services have motivated a fresh look at the design of data center network infrastructure in multiple layers. To transfer the huge amount of data generated by many data intensive applications, data center network has to be fast, scalable and power efficient. To support flexible and efficient sharing in cloud services, service providers deploy a virtualization layer as part of the data center infrastructure.

This thesis explores the design and performance analysis of data center network infrastructure in both physical network and virtualization layer. On the physical network design front, we present a hybrid packet/circuit switched network architecture which uses circuit switched optics to augment traditional packet-switched Ethernet in modern data centers. we show that this technique has substantial potential to improve bisection bandwidth and application performance in a cost-effective manner. To push the adoption of optical circuits in real cloud data centers, we further explore and address the circuit control issues in shared data center environments. On the virtualization layer, we present an analytical study on the network performance of virtualized data centers. Using Amazon EC2 as an experiment platform, we quantify the impact of virtualization on network performance in commercial cloud. Our findings provide valuable insights to both cloud users in moving legacy application into cloud and service providers in improving the virtualization infrastructure to support better cloud services.

Acknowledgments

I would like to express my gratitude to all those who have guided, helped and supported me on the path toward where I am today. Without them, this thesis would not be possible.

My heartfelt thanks go to my advisor Prof. T. S. Eugene Ng. Eugene helped me in all the time of my research. His insights, stimulating suggestions and high standard helped me improve the quality of my research work and shaped my way of thinking in research. He also taught me in so many ways, such as how to handle myself when I am lost in research and in life. His helps and support were essential in my completion of this thesis.

I want to thank my internship mentors Dina Papagiannaki and Michael Kaminsky at Intel Labs Pittsburgh. Dina and Michael provided me such wonderful experiences at Intel labs and gave me the opportunity to connect to much broader research community outside Rice. I also thank them for serving in my thesis committee and give tremendous helps to my thesis research.

A large part of my research work was collaborated with Prof. David Andersen at Carnegie Mellon University, Michael Kozuch and Michael Ryan at Intel Labs Pittsburgh, George Porter, Hamid Bazzaz, Malveeka Tewari and Prof. Amin Vahdat at University of California, San Diego. They not only help me with access to testbed infrastructure at Intel labs and UCSD. Their insightful feedbacks also elevate my research work to a new level. The discussion with them helped a lot in my research and made our collaboration the most productive part.

The system group at Rice gave me many productive discussions. Zheng Cai, Florin Dinu, Bo Zhang and Jie Zheng gave me so many good suggestions on my research and presentations. Several of my research ideas were fleshed out in invaluable discussions with Yanjun Sun, and my office mates, Scott Crosby, Shu Du and Kaushik Ram.

I also want to thank our department coordinator, Bel Martinez. Bel effectively helped me meet deadlines, fill out forms appropriately, prepare for my defense, and much more. Her helps smoothly navigate me through the administrative procedure.

Graduate school is stressful. I would never have gotten to this point without the support from my family and friends. My parents have always trusted my choices and stood on my side in the long time. Angela, Bo, Donghua, Florin, Jie, Ruiqiang, Yi and many other friends at Rice have made my graduate study fulfilled and memorable.

Contents

Illustrations

Chapter 1

Introduction

1.1 Data Centers for Cloud Services

Today, we are living in a "big data" era. Modern computer applications generate a huge amount of data everyday. For example, a single scientific telescope can generate more than 200GB of images each day; Ebay has more than 6.5 peta-bytes of business data about its customers; Youtube transmitted more than 27 Peta-bytes of video streams to millions of users every month in 2006; Google caches the whole World Wide Web and has to process more than 20 Peta-bytes of data per day to provide the Internet search service. The Internet search giant has to build exa-byte level storage systems to prepare for the explosion of Internet data in the near future.

Many companies and organizations build large data centers to store and process the huge amount of data for various applications. For example, both Google and Microsoft have been reported to have tens of data centers built world wide. These big data centers contains hundreds of thousands of servers, which are organized into thousands of server racks and connected by a high speed network. The servers in data centers are organized to support various distributed applications, such as Internet services, scientific computing and parallel data processing.

The emerging cloud service business [AFG+09] opens the enterprise data centers to individual and enterprise users. In a cloud service, a service provider will share its data centers and let users lease computing and storage resources using the pay-as-you-go model. There are several business models for cloud services, such as Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). These different service models refer to applications, system software or hardware infrastructure delivered as ser-

vices over the Internet. The data center hardware and software built to support these cloud services is what people call a *cloud.*

Using cloud services, individual and small business users do not need to build and operate their own IT infrastructure to deploy their innovative ideas for new Internet services. Instead, they can rent them from service providers on demand. The on-demand nature of this service allows users to not worry about the over-provisioning of computing resources for a service whose popularity does not meet their predictions, or under-provisioning for one that becomes wildly popular. It provides a flexible and cost effective way of IT service and resource sharing. Since its inception, the cloud service model has attracted a lot of attention from both users and service providers. Currently, several big vendors have a cloud service business, for example, Amazon EC2, Google AppEngine and Microsoft Azure.

1.2 Data Center Network Infrastructure: Challenges and State of the Art

To support Internet cloud service, service providers have to build data center infrastructure that can store and process large amount of data, manage and handle requests from a large number of users and support various types of applications with strict performance requirements. It brings grand challenges to the design of data center infrastructure on every aspect. This thesis is mainly focused on the the challenges on networking infrastructure in cloud data centers. Previous studies [AFG+09] have discussed many other challenges in designing a large scale data center system for cloud services, such as service availability and reliability, network security, and performance predictability. These issues are out of scope of this thesis.

1.2.1 Bandwidth Bottleneck

Many data center applications are data intensive. Distributed applications, such as MapReduce [DG04b] shuffle large amount of data among all the selected servers, which require

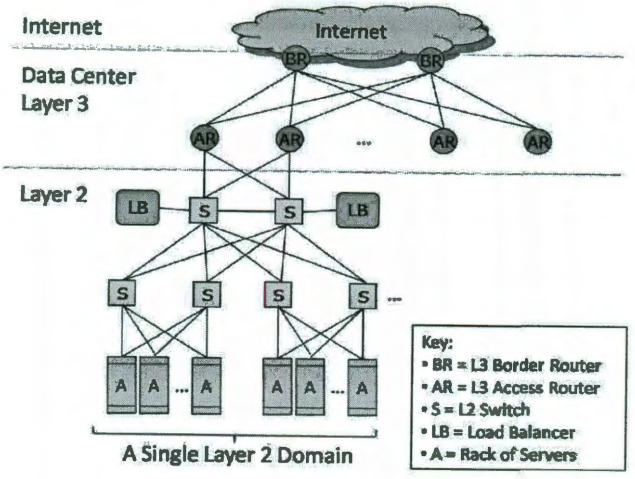


Figure 1.1 : Conventional Tree Structure Data Center Network (figure adopted from [Sys08])

very high bandwidth across different server racks. Figure 1.1 shows the hierarchical Ethernet network architecture in conventional data centers, where 10–40 servers are placed in a server rack with a "Top of Rack" (ToR) switch in each. The ToR switches are the leaves in a tree of Ethernet switches that connects all of the racks. The bandwidth at the top of the tree is typically a fraction of the incoming capacity, creating a bottleneck.

Traffic will be congested at the top layer of tree structure network when applications send large amount of data across different server racks. So the first challenge in the data center network is the bandwidth bottleneck problem.

To remove the bandwidth bottleneck in data center networks, Research community has begun exploring novel interconnect topologies to provide high bisection bandwidth using commodity Ethernet switches—examples include Fat trees [AFLV08, MPF+09, GJK+09], DCell [GWT+08a], and BCube [GLL+09], among a rapidly growing set of alternatives, many adapted from earlier solutions from the telecom and supercomputing areas. These new designs provide optimal switching capacity, but they require a large number of links and switches. For example, a k-level fat tree used to connect N servers needs at least $N \times k$ switch ports and wires. While research is ongoing in this area, physically constructing these

topologies at present requires complex, structured wiring, and expanding the networks after construction is challenging.

1.2.2 Flexible Control and Efficient Sharing

Cloud data center is a shared infrastructure among a large amount of users. Users may run various applications in the data center with different performance requirements. So another challenge for service providers is to configure and control the data center network and servers, and share them efficiently among different users and applications.

The major difficulty in data center sharing is to achieve the cost-effective sharing but still preserve performance isolation among different applications. Data center operators will have to build control frameworks that can implement flexible control policies among various users.

Most cloud service providers use machine virtualization techniques to provide flexible and cost-effective resource sharing among users. For example, both Amazon EC2 [Ama] and GoGrid [GoG] use Xen virtualization [BDF+03] to support multiple virtual machine instances on a single physical server.

With the virtualization technology, service providers can install different guest OSes on a physical server. They can also dynamically provision and de-provision hardware resources to virtual machines based on the demand of users, which allows flexible and efficient sharing of physical resources in data centers. Applications running in different virtual machine instances will be consolidated in different execution environment, therefore providing good isolation among different cloud users. Virtualization provides a flexible way of sharing processor, memory and storage resources in cloud data centers.

However, virtualization of I/O and network is more tricky. It is expected that virtualization can impact the communication performance in cloud data centers because virtual machine instances are sharing the buffers and forwarding paths in the network. But very few studies have been performed to understand the characteristics of these large scale virtualized cloud environments. With users considering moving more and more applications

into cloud, it is important to understand the networking performance and its application layer implications in virtualized cloud data centers.

1.3 Thesis Contributions

This thesis is focused on the design and performance analysis of data center network infrastructure for cloud services. We explore the design and performance issues of cloud data centers in both physical network and virtualization layers.

The design of a hybrid data center network architecture: For the physical network architecture aspect, we propose a Hybrid Packet and Circuit switched (HyPaC) data center network to provide high bandwidth for data intensive applications with low complexity. The HyPaC network augments the traditional hierarchy of Ethernet with a simple rack-to-rack optical circuit switched network. The optical network is reconfigured relatively slowly compared to the per-packet electrical switches, connecting at any point in time each rack to exactly one other rack. As a result, pairs of racks experience transient high capacity links, and the set of racks that are paired changes over time based upon traffic demand.

Among numerous design choices that realize the HyPaC architecture, we develop a prototype system called c-Through in which the responsibility for traffic demand estimation and traffic demultiplexing resides in end hosts, making it compatible with existing packet switches and transparent to data center applications. Experiments with a c-Through prototype on an emulated testbed show that optical circuits can be integrated efficiently with traditional Ethernet and TCP/IP based protocols without modifying applications and Ethernet switches. By experimenting with different kinds of applications over the c-Through prototype, we provide insight into how the HyPaC architecture applies to several usage scenarios. We find that HyPaC can benefit many kinds of applications, but particularly those with bulk transfer components, skewed traffic patterns, and loose synchronization.

We further explore the control of optical circuit in heterogeneous data centers. We identify a set of the challenges of adopting optical circuit switches in a cloud data center with non-cooperative applications sharing the optical circuits. To address these challenges,

we propose a "observe-analyze-act" control framework that can monitor data center traffic from multiple sources and realize more flexible control policies among different applications using OpenFlow switches. We discuss the algorithms to address the circuit configuration issue with more application semantics.

Network performance analysis of virtualized data centers: On the virtualization aspect, we perform a first analytical study of the networking performance of virtualized data centers. We present an empirical measurement study on the end-to-end networking performance of the commercial Amazon EC2 cloud service, which represents a typical large scale data center with machine virtualization. The focus of our study is to characterize the networking performance of virtual machine instances and understand the impact of virtualization on the network performance experienced by users.

We observe wide-spread processor sharing on small instances of Amazon EC2. We find that processor sharing and I/O sharing can cause unstable TCP/UDP throughput and abnormally large packet delay variations among Amazon EC2 instances. The abnormally unstable network performance can dramatically skew the results of certain network performance measurement techniques. Our study provide first hand insights on end-to-end networking performance in virtualized clouds, which are valuable to both cloud users and service providers. We discuss the implications of our findings on various cloud applications and the design of a virtualization infrastructure in cloud data centers.

The rest of this thesis are organized as follows: Chapter 2 discusses the related work in literature on data center network infrastructures. Chapter 3 presents the hybrid packet/circuit switched data center network architecture and its basic design and performance study. Chapter 4 discusses the control of optical circuits in heterogeneous data centers. Chapter 5 analyzes the networking performance in virtualized data centers. We discuss the future work in Chapter 6 and conclude the thesis in Chapter 7.

Chapter 2

Background and Related Work

2.1 Data Center Network Architectures

The first tide of data center network architecture research are focused on building scalable and high bisection bandwidth network using commodity packet switches. Previous studies [AFLV08, MPF+09, GJK+09] propose data center network architectures using Fat-Tree topologies. To construct a large scale layer 2 domain in data center, they proposals leverage address translation or tunneling techniques to construct a large scale layer 2 domain without the need of flooding as in Ethernet. Another line of studies, such as DCell [GWT+08b], BCube [GLL+09] and CamCube [HALOD10], are focused server centric network architectures in data centers, which connect servers directly using variations of HyperCube topology and leverage servers as intermediate hops in communications. These network architectures can provide good load balancing routing and fault tolerance, but they require additional processor and network interfaces on servers for the intermediate forwarding tasks. All these proposals achieve high bandwidth using commodity packet switch components, which requires large numbers of cables and devices and increases the network complexity. On the protocol design aspect, Seattle [KCR08] is a system design that improves the scalability of traditional Ethernet using the ideas borrowed from DHT and layer 3 link state routing protocols.

2.2 Hybrid Electrical/Optical Network Design

In addition to the aforementioned data center topologies (FatTrees, DCell, BCube, etc.), two recent studies on data center networking address parts of our problem domain. Our

previous papers [GAKM09, WAK+09] proposed the basic ideas of using optical circuits to augment an under-provisioned packet-switched network in data centers. Two recent proposals sketched designs that are similar in spirit to ours. One observed that many applications that run in their production datacenters do not require full bisection bandwidth. They instead proposed the use of 60GHz wireless "flyways" to augment an electrical network provisioned for average-case use, using the wireless links to selectively add capacity where needed [KPB09]. The second one is parallel work Helios [FPR+10], which explores a similar hybrid electrical/optical data center architecture. A key difference between Helios and c-Through is that Helios implements its traffic estimation and traffic demultiplexing features on switches. This approach makes traffic control transparent to end-hosts, but it requires modifying all the switches. An advantage of the c-Through design is that by buffering data in the hosts, c-Through can batch traffic and fill the optical link effectively when it is available. Helios and c-Through demonstrate different design points and performance trade-offs in the hybrid data center design space.

The supercomputing community has also extensively examined the use of optical circuits, though their goals often differ substantially from our focus. Within a supercomputer, several papers examined the use of node-to-node circuit switched optics [BBea05]; in contrast, our work deliberately amortizes the potentially high cost of optical ports and transceivers by providing only rack-to-rack connectivity, a design we feel more appropriate for a commodity datacenter. Our work by necessity then focuses more on the application and operating systems challenges of effectively harnessing this restricted pattern of communication. IBM researchers explored the use of hybrid networks in a stream computing system [SZW+09]. While it provides no design details, this work focused primarily routing and job management in stream computing.

Using large per-destination queues at the edge to aggregate traffic and make use of optical paths is related to optical burst switching [QY99] in optical networks. Many research efforts examined the use of optical burst switching in the Internet backbone (e.g [Tur99, YQD00a]), but they focus mostly on distributed scheduling and contention reso-

lution in order to correctly integrate the optical paths. Similarly, UCLP (User Controlled Lightpaths) [wwwj] and numerous other technologies (e.g., MP-λ-S) switch optical circuits on hour-and-longer timescales for wide-area connectivity and computing. In contrast, our datacenter focus limits the amount of traffic available to statistically multiplex onto the optical links, but simultaneously grants the flexibility to induce traffic skew and to incorporate the end-hosts into the circuit switched network. Our results suggest that the increased control and information from this integration substantially improves the throughput gains from optical links.

2.3 Circuit Control in Shared Data Centers

Recent studies have explored the problem of sharing datacenter network among multiple tenants. SecondNet [GLW+10] is a virtual datacenter architecture that controls the bandwidth allocation to applications with different service types in hypervisors. Seawall [SKGK11] allocates network bandwidth among non-cooperative applications using congestion controlled, edge to edge tunnels. These studies are mostly focused on network sharing in virtualized datacenter environment, while our work targets specifically on HyPaC network and we explicitly address the optical circuit allocation problem for mixed applications.

Optical circuit scheduling has been studied extensively in the context of backbone network. For example, previous work [YQD00b, YQD01] have studied the circuit scheduling with QoS guarantee in optical burst switching network. In more recent work [And09], Andrei et al. have studied the provisioning of data intensive applications over optical backbone network. Another work [DPS+10] proposes a unified control plane for IP/Ethernet and optical circuit switched network using OpenFlow. Our work is different from these existing studies because datacenter environment is very different from traditional backbone network. Applications are different, traffic patterns are different and in datacenter, the optical circuits must be reconfigured much more frequently to accommodate the dynamics in application traffic. Therefore, the circuit allocation mechanism in datacenters must be able

to adaptive to traffic changes more quickly.

Another line of related work is recent proposals to control datacenter networks with a software controller with global view of the topology [GHM+05, CGA+06, MFP+07, GKP+08]. The idea is to provide a high level logically centralized (potentially physically replicated) software component to observe and control a network. The main advantages of this is flexibility as the network programs/protocols would be written as if the entire network were present on a single decision element as opposed to requiring a new distributed algorithm across all network switching elements. Also that programs may be written in terms of high-level abstractions such as user, host names, not low level configuration parameters (e.g., IP and MAC addresses).

2.4 Virtualization and Cloud Network Performance

A few studies have evaluated the performance of cloud services. In [Gar07], Garfinkel has reported his experience of using Amazon EC2 and S3 (simple storage service) services. The focus is mainly on the performance of the Amazon S3 service. This paper measures the throughput and latency of the S3 service from Amazon EC2 and other Internet locations. In contrast, the focus of our study is on the networking performance of Amazon EC2 instances and the impact of virtualization on the network performance. In [Wal08], Walker evaluates the performance of Amazon EC2 for high-performance scientific computing applications. The author compares the performance of Amazon EC2 against another high performance computing cluster NCSA, and reports that Amazon EC2 has much worse performance than traditional scientific clusters. This paper is focused on the application level performance of Amazon EC2. The findings of our study can help to explain some of the observations in [Wal08].

In [EYD07], Ersoz *et al.* measure the network traffic characteristics in a cluster-based, multi-tier data center. This study is not based on a real commercial data center service. Instead, the authors measure the network traffic using an emulated data center prototype. A more recent study [BAAZ09a] has measured the network traffic workload of production

data centers. The focus of that work is to investigate the data center traffic pattern and explore the opportunities for traffic engineering. No virtualization is considered in these studies. As far as we know, our work is the first study focusing on the networking performance of Amazon EC2 instances and on understanding the impact of virtualization on the data center network performance.

Several studies [MST+05, MCZ06] have evaluated the overheads of Xen network virtualization and proposed new techniques to improve the network performance of Xen virtual machine. These studies are all performed on two directly connected Xen virtual machines. The focus is try to optimize their network throughput from the operating system's perspective. In [OCR08], Ongaro *et al.* studied the impact of Xen scheduler on the I/O performance using multiple guest domains concurrently running different types of applications. This study is performed on a controlled Xen testbed and the observations are well aligned with our findings on the I/O bandwidth and latency instability. This paper also discusses a few enhancement techniques on Xen scheduler to reduce the impact on I/O performance.

Chapter 3

Hybrid Packet/Circuit Switched Data Center Network

3.1 Motivation and Solution Outline

The rising tide of data-intensive, massive scale cluster computing is creating new challenges for datacenter networks. In this chapter, we explore the question of integrating optical circuit-switched technologies into this traditionally packet-switched environment to create a "HyPaC network"—Hybrid Packet and Circuit—asking both *how* and *when* such an approach might prove viable. We ask this question in the hope of being able to identify a solution that combines the best of both worlds, exploiting the differing characteristics of optical and electrical switching: optics provides higher bandwidth, but suffers slower switching speed. Our results suggest in particular that data-intensive workloads such as those generated by MapReduce, Hadoop, or Dryad are sufficiently latency-insensitive that much of their traffic can be carried on slowly-switching paths. Our results both raise many questions for future work regarding the design details for hybrid networks, and motivate the need to answer those questions by showing that such designs are feasible, exploring the application characteristics that render them so, and providing a first-cut design to exploit them.

To understand these goals, first consider today's hierarchical, electrically-switched datacenter networks. They typically place 10–40 servers in a rack, with an aggregation ("Top of Rack", or ToR) switch in each. The ToR switches are the leaves in a tree of Ethernet switches that connects all of the racks. The bandwidth at the top of the tree is typically a fraction of the incoming capacity, creating a bottleneck. In response to this now-well-known limitation, the research community has begun exploring novel interconnect topologies to provide high bisection bandwidth using commodity Ethernet

switches—examples include Fat trees [AFLV08, MPF+09, GJK+09], DCell [GWT+08b], and BCube [GLL+09], among a rapidly growing set of alternatives, many adapted from earlier solutions from the telecom and supercomputing areas. These new designs provide optimal switching capacity, but they require a large number of links and switches. For example, a k-level fat tree used to connect N servers needs at least $N \times k$ switch ports and wires. While research is ongoing in this area, physically constructing these topologies at present requires complex, structured wiring, and expanding the networks after construction is challenging.

For many years, optical circuit switching has led electrical packet switching in high bandwidth transmission. A single optical fiber can carry hundreds of gigabits per second. However, this capacity must be allocated between a source and a destination at coarse granularity—much longer than the duration of a single packet transmission. "All-optical" packet switched networks have been a goal as elusive as they are important; despite numerous innovations, today's commodity optical switching technologies require on the order of milliseconds to establish a new circuit. They provide high bandwidth, but cannot provide full bisection bandwidth at the packet granularity.

Why, therefore, do we believe optical circuit switching is worth considering in data centers? Because the characteristics of many data-centric workloads suggest that many data center applications may not need full bisection bandwidth at the *packet granularity*. Recent measurement studies show substantial traffic "concentration" in data center workloads: In many scientific computing applications *"the bulk of inter-processor communication was bounded in degree and changed very slowly"* [BBea05]. Microsoft researchers measured the traffic characteristics of production data centers, finding *"evidence of ON-OFF traffic behavior"* [BAAZ09b] and *"only a few ToRs are hot and most of their traffic goes to a few other ToRs"* [KPB09]. As we expand upon later, these patterns and others require high bandwidth, but the concentration of traffic makes it more suited to a network in which at any time, only a subset of the paths are accelerated.

In this work, our goal is to develop an architecture that can exploit these more funda-

mental differences between packet-switched electrical networks and circuit-switched optical networks (or, more generally, any circuit-switched technology with a bandwidth advantage) in the context of a modern datacenter. We only give a brief discussion of the relative cost of these networks: we believe it is a question that follows those that we ask about feasibility and the technical requirements to their use.[1] Parallel work [FPR+10] provides a more detailed discussion on the cost issue. We present a first effort exploring this design space and makes the following contributions:

(1) We present a hybrid packet and circuit switched data center network architecture (or HyPaC for short) that augments a traditional electrical packet switch hierarchy with a second, high-speed *rack-to-rack* circuit-switched optical network. The optical network is reconfigured relatively slowly compared to the per-packet electrical switches, connecting at any point in time each rack to exactly one other rack. As a result, pairs of racks experience transient high capacity links, and the set of racks that are paired changes over time based upon traffic demand. Among numerous design choices that realize the HyPaC architecture, we present a prototype system called c-Through in which the responsibility for traffic demand estimation and traffic demultiplexing resides in end hosts, making it compatible with existing packet switches. In order to make the best use of the transient high capacity optical circuits, c-Through recruits servers to buffer traffic so as to collect sufficient volumes for high speed transmission. Traffic buffering is done by enlarging individual socket buffer limits so as to do it without introducing head-of-line blocking or extra delay. By performing this buffering in-kernel, we explore the benefits of the HyPaC architecture without substantial application modification. Experiments with a c-Through prototype on an emulated testbed show that optical circuits can be integrated efficiently with traditional Ethernet and TCP/IP based protocols. While we emphasize that there are many other ways that optical circuit switching might be used to enhance data center networks, the c-Through design shows that the general approach is feasible, even without modifying applications or

[1]This question also falls outside our bailiwick: Price is very sensitive to volume and market conditions that may change drastically if optical technologies were to become widely deployed in the datacenter.

Ethernet switches.

(2) By experimenting with different kinds of applications over the c-Through prototype, we provide insight into how the HyPaC architecture applies to several usage scenarios. We find that HyPaC can benefit many kinds of applications, but particularly those with bulk transfer components, skewed traffic patterns, and loose synchronization. We provide guidelines on how to maximize the benefits of the HyPaC architecture in large scale data centers.

3.2 Optics: Pro and Con

Optical circuit switching can provide substantially higher bandwidth than electrical packet switching. Today's fastest switches and routers are limited to roughly 40Gb/s per port; in contrast, 100Gb/s optical links have been developed [wwwh], and WDM techniques can multiplex terabits/s onto a single fiber in the lab [wwwa]. Today's market already offers 320x320 optical circuit switches with 40Gb/s transceivers [wwwb]. The cost it pays is requiring about 20ms to switch to a new mapping of input ports to output ports. In contrast, the CRS-1 router from Cisco supports only sixteen 40Gb/s line cards in a full-rack unit—but, of course, provides packet-granularity switching.

Slow switching is a lasting challenge for optical networking, and affects all commercially available technologies. MEMS (Micro-Electro-Mechanical Systems) optical switches reconfigure by physically rotating mirror arrays that redirect carrier laser beams to create connections between input and output ports. The *reconfiguration time* for such devices is a few milliseconds.[2] Tunable lasers combined with an Arrayed Waveguide Grating Router (AWGR) can potentially provide faster switching. Tunable lasers can switch channels in tens of nanoseconds, with a significant caveat: "dark tuning". To avoid spilling garbage into the network during switching, the laser must be optically isolated from the network. With the practical constraints involved, the best switching speeds available today

[2] e.g., opneti's 1x8 MEMS switch requires 2ms typical, 5ms max to switch with multi-mode fiber. `http://www.opneti.com/right/1x8switch.htm`

are still in the 1 to 10ms range [BM06]. Tunable lasers will likely switch more rapidly in the future, which would likely improve the performance provided by a system such as c-Through, but, of course, such improvements over time must also be balanced against the increased number of packets per second transmitted on the links.

Optics has traditionally been viewed as more expensive than its electrical counterparts, but this gap has been narrowing over time, particularly as newer Ethernet standards require increasingly heavy and expensive cables (such as CX4) for high bandwidth over even modest distances. For example, the price of optical transceivers has dropped more than 90 percent in the last decade [Jos09], and the price of MEMS optical switches has dropped to a few hundred dollars per port [www03]. Even at recent prices, the cost of optical networking components is comparable with existing solutions. For example, it has been estimated that constructing a BCube with 2048 servers costs around $92k for switches and NICs and requires 8192 wires [GLL+09]. Today, MEMS switches are mostly aimed at the low-volume, high-margin test and measurement market, but even so, using a MEMS switch to connect 52 48-port switches, each switch connecting 40 servers, would cost approximately $110k at most (On an 80-port MEMS optical switch, each port costs $200-$700, single 10Gbit optical transceiver modules cost under $350, and 48-port switches cost under $700). We expect the cost of these switches would drop substantially were they to be used in commodity settings, and the cost of the transceivers drops continuously. The increasing bandwidth demand and dropping prices of optical devices make optical circuits a viable choice for high bandwidth transmission in data centers.

3.3 Feasibility Analysis

Figure 3.1 depicts the HyPaC configuration we use in the rest of this thesis: The packet-switched network (top) uses a traditional hierarchy of Ethernet switches arranged in a tree. The circuit-switched network (bottom) connects the top-of-rack switches. Optically connecting racks instead of nodes reduces the number of (still expensive) optical components required, but can potentially still provide high capacity because a single optical path can

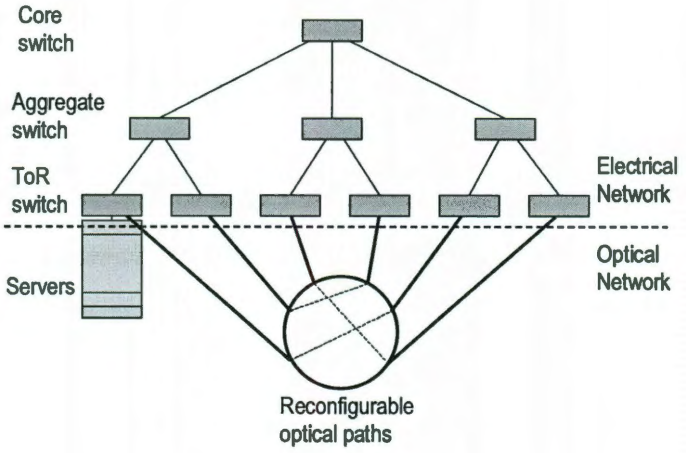


Figure 3.1 : HyPaC network architecture

handle tens of servers sending at full capacity over conventional gigabit Ethernet links.

The circuit-switched network can only provide a *matching* on the graph of racks: Each rack can have at most one high-bandwidth connection to another rack at a time. The switch can be reconfigured to match different racks at a later time; as noted earlier, this reconfiguration takes a few milliseconds, during which time the fast paths are unusable. To ensure that latency sensitive applications can make progress, HyPaC retains the packet-switched network. Any node can therefore talk to any other node at any time over potentially oversubscribed packet-switched links.

For the circuits to provide benefits, the traffic must be "pairwise concentrated"—there must exist pairs of racks with high bandwidth demands between them and lower demand to others. Fortunately, such concentration has been observed by numerous prior studies [BBea05, BAAZ09b, KPB09]. This concentration exists for several reasons: time-varying traffic, biased distributions, and—our focus in later sections—amenability to batching. First, applications whose traffic demands vary over time (e.g. hitting other bottlenecks, multi-phase operation) can contribute to a non-uniform traffic matrix. Second, other applications have intrinsic communication skew in which most nodes only communicate with a small number of partners. This limited out-degree leads to concentrated communication.

Finally, latency-insensitive applications such as MapReduce-style computations may be amenable to batched data delivery: instead of sending data to destinations in a fine-grained manner (e.g., 1, 2, 3, 2, 3, 1, 2), sufficient buffering can be provided to batch this delivery (1, 1, 2, 2, 2, 3, 3). These patterns do not require arbitrary full-bisection capacity.

The feasibility of our proposal depends on the agility of optical path reconfiguration and the existence (or potential to induce) traffic skew. In this chapter, we perform an analytical study on the feasibility of using circuit-switched optical paths in data centers. We discuss the algorithm that can be used to compute the configuration of optical paths based on the traffic demands. We estimate the factors that impact the optical path reconfiguration time. We study the workloads of an operational data center to estimate the amount of traffics that can be taken by optical paths.

3.3.1 Optical Reconfiguration Algorithm

Suppose the cross-rack traffic matrix is given (we discuss later how to estimate it), we need to figure out how to connect the server racks by optical paths in order to maximize the amount of traffic offloaded to the optical network. This can be formulated as a maximum weight perfect matching problem. The cross rack traffic matrix is a graph $G = (E, V)$. V is the vertex set in which each vertex represents one rack and E is the edge set. The weight of an edge e, $w(e)$, is the traffic volume between the end vertices. A *matching* M in G is a set of pairwise non-adjacent edges. That is, no two edges share a common vertex. A *perfect matching* is a matching that matches all vertices of the graph. From this formulation, the optical configuration is a perfect matching with the maximum aggregated weight. The solution can be computed in polynomial time by Edmonds' algorithm [Edm65].

3.3.2 Optical Path Reconfiguration Time

The ability of the optical network to relieve bottlenecks in the electrical network will depend on its agility to reconfigure and accommodate varying traffic demands. There are several factors that influence how often one could reconfigure the optical network in our

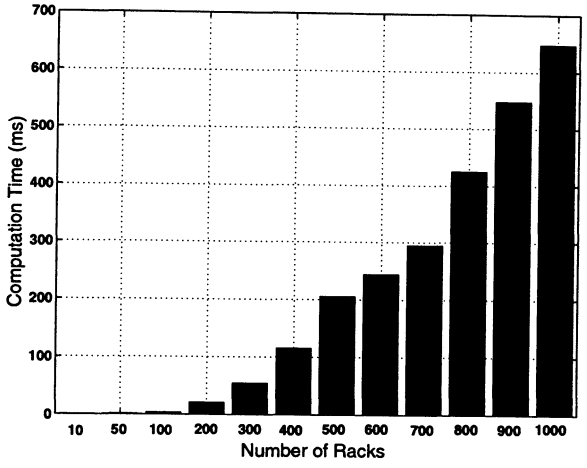


Figure 3.2 : Optical Path Computation Time

approach. First, there is the circuit setup signaling delay, which, for data center networks, should be small ($< 1ms$). Second, the setup of optical paths implies the physical manipulation of mirrors that have specific hardware switching times. During this fixed period, the optical paths are down. For MEMS-based optical switches, the hardware reconfiguration time is a few milliseconds. Third, the reconfiguration interval is also lower bounded by the time required to compute the rack-to-rack optical path configuration, which depends on the network size.

We use the Blossom VI implementation [CR99] of Edmonds' algorithm to compute the optical path configuration. Figure 3.2 shows the configuration computation time for random rack matrices with different numbers of racks. For each rack matrix size, we generate 5 random matrices and present the average computation time versus rack size in Figure 3.2. The computation runs on one core of an Intel Xeon 3.2GHz processor with 2GB of memory. The results show that the optimal configuration can be computed rapidly—around 640 ms for 1000 racks. Consequently, even in very large data centers, one could envision the reconfiguration of the optical network happening at relatively small time scales, thus able to accommodate varying traffic demands.

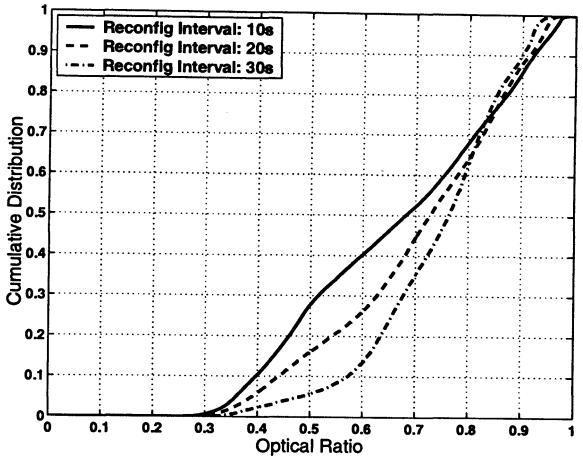


Figure 3.3 : The Cumulative Distribution of Optical Ratios for One-week Data Center Traffic

3.3.3 Evidence for Traffic Skew

Augmenting a traditional data center architecture with a reconfigurable circuit-switched optical network introduces additional capacity that may relieve some of its inherent bottlenecks. However, this holds true only if the rack-to-rack traffic is, or can be made, skewed, and if this skew can be identified and exploited appropriately. We study the workload of one small operational data center to assess the properties of today's data center workloads (even though we admit that our results could not generalize across all possible data centers).

Our analysis is based on a small seven-rack research datacenter with a total of 155 servers and 1060 cores. We instrument all servers with the IPTables NetFlow module and export data every 10 seconds. We then aggregate this server-to-server traffic matrix into a rack-to-rack traffic matrix based on the datacenter topology. The traffic captured includes a variety of workloads, such as MPI, Hadoop, and scientific computing applications.

We input the rack-to-rack traffic matrix to Edmonds' algorithm for the perfect weighted matching that will identify the 3 rack-to-rack flows that can be routed on top of the optical network (the 7 racks allow for 3 non-overlapping rack-to-rack connections). The sum of the volume of those 3 flows represents the maximum amount of traffic that could be offloaded

from the electrical network onto the optical paths. We analyze a one-week traffic trace using three different optical path reconfiguration intervals.

We define the optical ratio to capture the total traffic volume of those 3 rack-to-rack optical flows over the overall cross-rack traffic in the data center. We then plot its empirical cumulative distribution function in Figure 3.3. Note that if rack-to-rack traffic is uniform, the fraction of offloaded traffic would be $3/21 = 14\%$ (out of the 21 rack-to-rack flows, only 3 can be routed optically).

Instead, Figure 3.3 shows that setting up 3 optical paths between the 7 racks in the data center can offload more than 50% of the total cross rack traffic in most cases. Reconfiguring the optical network every 30 seconds results in higher fractions of the overall traffic routed optically, taking advantage of increased aggregation. This result demonstrates that using even a few optical paths has the potential to offload significant amounts of traffic from the electrical network. Further study is needed to derive the best optical reconfiguration time based on the dynamic traffic demands.

3.4 HyPaC Network Requirements

3.4.1 System Requirement

Table 3.1 summarizes functions needed for a generic HyPaC-style network. In the *control plane*, effective use of the circuit-switched paths requires determining rack-to-rack traffic demands and timely circuit reconfiguration to match these demands.

In the *data plane*, a HyPaC network has two properties: First, when a circuit is established between two racks, there exist two paths between them—the circuit-switched link and the always-present packet-switched path. Second, when the circuits are reconfigured, the network topology changes. Reconfiguration in a large data center causes hundreds of simultaneous link up/down events, a level of dynamism much higher than usually found in data centers. A HyPaC network therefore requires traffic control mechanisms to dynamically de-multiplex traffic onto the circuit or packet switched network, as appropriate.

	System requirements
Control plane	1. Estimating cross-rack traffic demands
	2. Managing circuit configuration
Data plane	1. De-multiplexing traffic in dual-path network
	2. Maximizing the utilization of circuits when available (optimization)

Table 3.1 : Fundamental requirements of HyPaC architecture.

Finally, if applications do not send traffic rapidly enough to fill the circuit-switched paths when they become available, a HyPaC design may need to implement additional mechanisms, such as extra batching, to allow them to do so.

3.4.2 Design Choices and Trade-offs

These system requirements can be achieved on either end-hosts or switches. For designs on end-hosts, the system components can be at different software layers (e.g, applications layer or kernel layer).

Traffic demand estimation: One simple choice is to let applications explicitly indicate their demands. Applications have the most accurate information about their demands, but this design requires modifying applications. As we discuss in Section 3.5, our c-Through design estimates traffic demand by increasing the per-connection socket buffer sizes and observing end-host buffer occupancy at runtime. This design requires additional kernel memory for buffering, but is transparent to applications and does not require switch changes. The Helios design [FPR+10], in contrast, estimates traffic demands at switches by borrow-

ing from Hedera [AFRR+10] an iterative algorithm to estimate traffic demands from flow information.

Traffic demultiplexing: Traditional Ethernet mechanisms handle multiple paths poorly. Spanning tree, for example, will block either the circuit-switched or the packet-switched network instead of allowing each to be used concurrently. The major design choice in traffic demultiplexing is between emerging link-layer routing protocols [MNZ04, KCR08, GJK+09, MPF+09, wwwf] and partition-based approaches that view the two networks as separate.

The advantage of a routing-based design is that, by treating the circuit and packet-switched networks as a single network, it operates transparently to hosts and applications. Its drawback is that it requires switch modification, and most existing routing protocols impose a relatively long convergence time when the topology changes. For example, in link state routing protocols, re-convergence following hundreds of simultaneous link changes could require seconds or even minutes [BR01]. To be viable, routing-based designs may require further work in rapidly converging routing protocols.

A second option, and the one we choose for c-Through, is to isolate the two networks and to de-multiplex traffic at either the end-hosts or at the ToR switches. We discuss our particular design choice further in Section 3.5. The advantage of separating the networks is that rapid circuit reconfiguration does not destabilize the packet-switched network. Its drawback is a potential increase in configuration complexity.

Circuit utilization optimizing, if necessary, can be similarly accomplished in several ways. An application-integrated approach could signal to applications to increase their transmission rate when the circuits are available; the application-transparent mechanism we choose for c-Through is to buffer additional data in TCP socket buffers, relying on TCP to ramp up quickly when bandwidth becomes available. Such buffering could also be accomplished in the ToR switches.

In the remainder of this chapter, we do not attempt to cover all of the possible design choices for constructing a HyPaC network. The following section introduces the c-Through

design, which represents one set of choices, and demonstrates that the HyPaC architecture can be feasibly implemented in today's datacenters without the need to modify switches or applications.

3.5 c-Through Design and Implementation

c-Through[3] is a HyPaC network design that recruits end-hosts to perform traffic monitoring, and uses a partition approach to separate the circuit (optical) and packet (electrical) networks. c-Through addresses all the architectural requirements outlined in Table 3.1. In our current testbed, we emulate a circuit switch's connectivity properties with a conventional packet switch. We therefore omit the implementation details on how the c-Through control software interfaces with a commercial circuit switch. However, existing interfaces for circuit configuration should be usable easily.

3.5.1 Managing Optical Paths

Traffic measurement: c-Through estimates rack-to-rack traffic demands in an application-transparent manner by increasing the per-connection socket buffer limit and observing per-connection buffer occupancy at runtime. This approach has two benefits: First, an application with a lot of data to send will fill its socket buffer, allowing us to identify paths with high demand. Second, as discussed below, it serves as the basis for optimizing use of the circuits.

The use of TCP socket buffers ensures that data is queued on a per-flow basis, thus avoiding head-of-line blocking between concurrent flows. A low-bandwidth, latency-sensitive control flow will therefore not experience high latency due to high-bandwidth data flows.

We buffer at end hosts, not switches, so that the system scales well with increasing node count (DRAM at the end hosts is relatively cheap and more available than on the ToR

[3] A conjunction of *c*, the speed of light, and "cut-through".

switches). Each server computes for each destination rack the total number of bytes waiting in socket buffers and reports these per-destination-rack demands to the optical manager.

Utilization optimization: Optical circuits take time to set up and tear down. c-Through buffers data in the hosts' socket buffers so that it can batch traffic and fill the optical link when it is available. As we show in Section 3.6, buffering a few tens to hundreds of megabytes of data at end hosts increases network utilization and application performance. The end-host TCP can ramp up to fill the increased available bandwidth quickly after the network has been reconfigured.

Optical configuration manager: The optical configuration manager collects traffic measurements, determines how optical paths should be configured, issues configuration directives to the switches, and informs hosts which paths are optically connected. The initial design of this component is a small, central manager attached to the optical switch (equivalent to a router control plane).

Given the cross-rack traffic matrix, the optical manager must determine how to connect the server racks by optical paths in order to maximize the amount of traffic offloaded to the optical network. This can be formulated as a maximum weight perfect matching problem. The cross rack traffic matrix is a graph $G = (E, V)$. V is the vertex set in which each vertex represents one rack and E is the edge set. The weight of an edge e, $w(e)$, is the traffic volume between the racks. A *matching* M in G is a set of pairwise non-adjacent edges. That is, no two edges share a common vertex. A *perfect matching* is a matching that matches all vertices of the graph. From this formulation, the optical configuration is a perfect matching with the maximum aggregated weight. The solution can be computed in polynomial time by Edmonds' algorithm [Edm65]. As we previously reported, Edmonds' algorithm is fast [WAK+09], computing the configuration of 1000 racks within a few hundred milliseconds.[4]

[4]At larger scales or if switching times drop drastically, faster but slightly heuristic algorithms such as iSLIP could be used [Mck99].

3.5.2 Traffic De-multiplexing

VLAN based network isolation: c-Through solves the traffic de-multiplexing problem by using *VLAN-based routing* (similar ideas using different mechanisms have been used to leverage multiple paths in Ethernet networks [SGNC04, ZWG09]). c-Through assigns ToR switches two different VLANs that logically isolate the optical network from the electrical network. VLAN-s handles packets destined for the packet-switched electrical network, and VLAN-c handles packets going directly to one other rack via the optical path. In c-Through, the end-host is responsible for tagging packets with the appropriate VLAN ID, for ease of deployment.

In this topology, observe that the topology of VLAN-s (packet-switched) does not change frequently, but that of VLAN-c could change a few times per second. Rapid reconfiguration is challenging for many Ethernet control protocols, such as spanning tree or other protocols with long convergence time [ECN06]. Therefore, spanning tree protocol should be disabled in VLAN-c and c-Through guarantees that the optical network is loop-free (by construction, it can provide only a matching). However, either existing or future protocols can still be used to manage the hierarchical electrical network.

Other designs are certainly viable depending on the technological constraints of the implementer: we do not believe that end-host VLAN selection is the only, or even the best, way to accomplish this goal in the long term. Our focus is more on demonstrating the fundamental feasibility of HyPaC networks for datacenters. Future advances in routing protocols and programmable switches (e.g., Click or OpenFlow) could provide a transparent, switch-based mechanism for traffic demultiplexing.

Traffic de-multiplexing on hosts: Each host runs a management daemon that informs the kernel about the inter-rack connectivity. The kernel then de-multiplexes traffic to the optical and electrical paths appropriately. As shown in Figure 3.4, each server controls its outgoing traffic using a per-rack output traffic scheduler. When TCP/IP transmits a packet, it goes to a destination-rack classifier, and is placed into a small (several packets) queue.

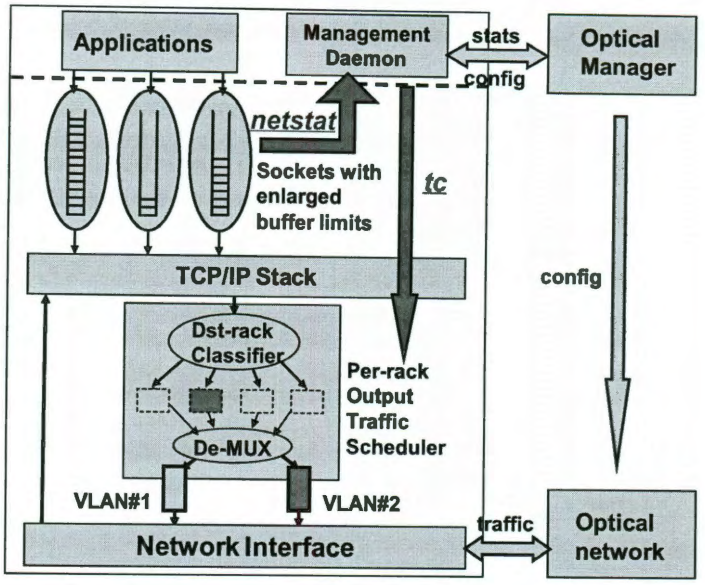


Figure 3.4 : The structure of the optical management system

Based upon their destination rack, the packets are then assigned either to the electrical or optical VLAN output queue (this queue is small—approximately the size of the Ethernet output buffer). Broadcast/multicast packets are always scheduled over the electrical network.

In this design, c-Through must give higher transmission priority to the optically-connected destinations than to the electrically-connected destinations. It does so using a weighted round-robin policy to assign weight 0.9 to the per-rack queue that is currently optically connected, and splits the remaining 0.1 weight among the non-optically-connected links. Because the policy is work-conserving and most traffic is running over TCP, this simple approach works well to ensure that both the optical and electrical networks are highly utilized and no flow is starved.

3.5.3 c-Through System Implementation

c-Through implements traffic measurement and control in-kernel to make the system transparent to applications and easy to deploy. Figure 3.4 shows the architecture of the optical

management system.

Two VLAN interfaces are configured on the physical NIC connected to the ToR switch. All packets sent through a VLAN interface are tagged with the associated VLAN number. As illustrated in Figure 3.4, packets going through interface VLAN#1 are tagged with VLAN#1, and forwarded via electrical paths by the ToR switch; packets sent out on interface VLAN#2 are tagged with VLAN#2 and forwarded via optical paths. Each server is configured with two virtual interfaces, bonded through a bonding driver. This approach is therefore transparent to the upper layers of the stack—the two virtual interfaces have the same MAC and IP addresses.

c-Through buffers traffic by enlarging the TCP socket buffer limits, allowing applications to push more data into the kernel. We use *netstat* to extract the buffered data sizes of all sockets and then sum them based on the destination rack.

In practice, the memory consumption from enlarged socket buffer limits is moderate. The buffers need not be huge in order to infer demand and to batch traffic for optical transfer: our results in Section 3.7 show that limiting the TCP send buffer to 100 MB provides good performance for many applications. Furthermore, the limit is not a lower bound: only as much memory is consumed as the application generates data to fill the socket buffer. The applications we observed do not generate unbounded amounts of outstanding data. For the data intensive applications we have tried, such as VM migration, MapReduce and MPI FFT, the total memory consumption of all socket buffers on each server rarely goes beyond 200MB.

A user-space management daemon on each node reads socket statistics using *netstat* and reports them to the central configuration manager. The statistics report how much traffic is buffered for each destination rack, permitting the optical configuration manager to assign an optimal configuration of the optical paths. To reduce the amount of traffic sent to the centralized manager, servers only report traffic statistics when the queue size is larger than a threshold and the queue size variation exceeds a second threshold. We empirically set both thresholds to 1MB. Since optical paths are configured for applications with high

bandwidth demands, omitting a small amount of buffered data will not significantly impact the configuration decision.

When a new configuration is computed, the manager sends reconfiguration commands to the optical switches and notifies the server daemons about the new configuration. The notification messages can be multicast to reduce overhead. The management daemon notifies the per-destination-rack output queue scheduler about the new optical path using *tc* (a Linux tool for configuring the kernel network stack). The scheduler then dequeues and de-multiplexes packets according to the new configuration. In a very large data center, the control traffic between individual servers and the central manager could be significant. We discuss this issue in Section 3.8.

3.6 System Evaluation

We implemented a c-Through prototype to study how well packet-switched and circuit-switched networks coexist and how applications perform. Due to the expense of prototyping a hybrid electrical/optical network, we emulate a HyPaC topology on a conventional packet-switched network, enabling and disabling communications to emulate the availability of high-speed optical links between switches. A controller introduces a reconfiguration delay to emulate the reconfiguration and settling time that the optical components would experience. Section 3.6.2 validates that this framework accurately reproduces network properties of interest. Although we choose parameters for these links and intervals based upon one particular optical technology (MEMS optical switches), our results should generalize as long as two assumptions continue to hold about optical and electrical networking: First, that the reconfiguration time of the optical switch remains long relative to the number of packets that can be sent (e.g., a 5ms reconfiguration interval is 16,000 packets of 1500 bytes on a 40 Gbps network). Second, that a circuit-switched optical path will retain a substantial bandwidth advantage over an electrically-switched path.

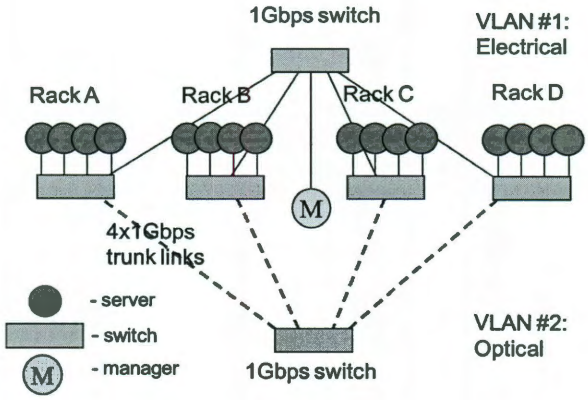


Figure 3.5 : The logical testbed topology.

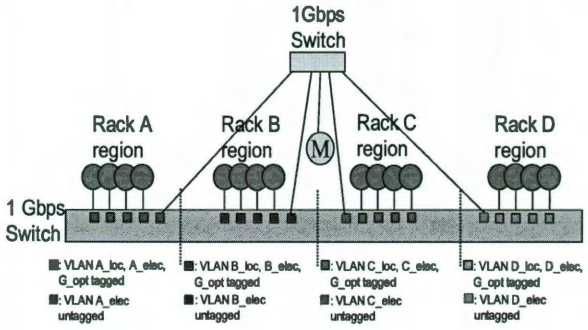


Figure 3.6 : The physical testbed configuration.

3.6.1 Testbed Setup

While there is no canonical data center design, one popular configuration includes 40 servers per rack connected at 1 Gbps to a ToR switch. Although a single rack can generate as much as 40 Gbps, the ToR switches are often interconnected at 1, 2, 4, or 10 Gbps, offering *over-subscription ratios* from 40:1 to 4:1 [wwwi, GJK+09].

We emulate comparable over-subscription ratios in our experiments. The logical topology emulated is shown in Figure 3.5. Physically, our emulation testbed consists of 16 servers and two Ethernet switches. Our switches are Dell PowerConnect 5448 Gigabit switches, and the servers have two 4-core Intel Xeon 2.5GHz processors and 8GB of mem-

ory. They run Ubuntu 8.04.3 LTS with the Linux 2.6.30 kernel. The 16 servers are connected to the first Gigabit Ethernet switch at 1 Gbps. We use VLANs to isolate the servers into four logical server racks as shown in Figure 3.6. The VLAN *X_loc* is used for intra-logical-rack traffic. Packets tagged with *X_loc* can only reach other servers within the same logical rack. Within each logical rack, one switch port is connected to the second Gigabit Ethernet switch (as shown in the top of Figure 3.6) which emulates a low speed packet-switched inter-rack network. By rate-limiting the ports, over-subscription ratios from 40:1 to 4:1 can be emulated. VLAN *X_elec* is used for traffic going through this low speed packet-switched inter-rack network. On the other hand, we emulate an optical circuit switch connecting the logical racks using the internal switching fabric of the first Ethernet switch. When the optical manager decides two logical racks are connected via an emulated circuit, communications through the emulated optical switch between these logical racks are allowed; otherwise, they are disallowed. An emulated optical circuit provides 4 Gbps of capacity between logical racks when communication is allowed. VLAN *G_opt* is used for traffic going through the emulated optical circuit-switched network.

Using an Ethernet switch to emulate the optical network creates a few differences between our testbed and a real optical network. First, we must artificially restrict communication through the emulated optical switch to emulate the limited rack-to-rack circuits. Second, we must make the network unavailable during optical switch reconfiguration. We estimate this delay based upon the switching time of MEMS switches plus a small settling time for the optical transceivers to re-synchronize before talking to a new destination. During this delay, no traffic is sent through the emulated optical switch.

For comparison, we also emulate a full bisection bandwidth packet-switched network in the testbed by allowing traffic to flow freely through the switching fabric of the first Gigabit Ethernet switch.

We implement the per-rack output scheduler in the kernel. The optical manager runs on a separate server. It collects traffic statistics from the rack servers and dynamically reconfigures the emulated optical paths. Because we use an Ethernet switch to emulate the

	FIFO		VLAN+output scheduling	
	Optical	Electrical	Optical	Electrical
TCP (Mbps)	940	94	921	94
UDP (Mbps)	945	94	945	94

Table 3.2 : Throughput with and without output scheduling.

optical switch, the optical manager does not command switches to set up optical paths. Instead, it communicates with the servers' management daemons to reconfigure the scheduler. When the manager starts to reconfigure optical paths, it first tells all servers to disable the optical paths by configuring the scheduler modules. It then delays for 10ms to emulate the switch reconfiguration, during which time no servers can use any optical path. The manager then instructs the servers to activate the new paths.

3.6.2 Micro-benchmark Evaluation

The goal of our micro-benchmarks is to understand how well today's TCP/IP stack can use the dynamic optical paths. In Section 3.7, we measure the benefit applications gain from the optical paths.

3.6.2.1 TCP behavior during optical path reconfiguration

We evaluate the effect of optical network emulation using electrical switches. We examine the TCP throughput between two servers in our testbed over several reconfiguration epochs. The servers transfer data as rapidly as possible from one to the other using the default TCP CUBIC. While the flow is running, we periodically set up and tear down the optical path between them. We emulate a 40:1 over-subscription ratio (100Mb/s between racks) to maximize the capacity change upon reconfiguration.

Figure 3.7 shows the TCP throughput at the receiver across the entire experiment, along with what that throughput would look like if the flow was routed on the electrical or the

optical path alone. The testbed correctly emulates the changes in network capacity before, during, and after reconfiguration. At time=1s, for instance, TCP correctly increases its throughput from 100 Mbps to 1 Gbps.

In this environment, TCP can increase its throughput to the available bandwidth within 5 ms of reconfiguration. When the optical path is torn down, TCP experiences packet loss since the available bandwidth drops rapidly. Although throughput briefly drops below the electrical network's capacity, it re-reaches the electrical capacity within a few tens of milliseconds. These experiments confirm that the optical emulation reflects the expected outcome of reconfiguration, and that TCP adapts rapidly to dynamically re-provisioned paths. This adaptation is possible because of the low RTT among servers. Even for a high bandwidth transmission, the TCP window size remains small.

3.6.2.2 How does the scheduler affect throughput?

The optical management system adds an output scheduler in the server kernel. The imposition of this component does not significantly affect TCP or UDP throughput. Table 3.2 shows the throughput achieved between a single sender and receiver using both 100 Mbps and 1 Gbps links (electrical and optical capacities, respectively), over TCP and UDP, and with and without the output scheduler. We gathered the results using *iperf* to send 800MB of data from memory as rapidly as possible.

3.6.2.3 Do large buffers affect packet delays?

c-Through is designed to avoid head-of-line (HoL) blocking by using per-flow socket buffers to buffer traffic, preventing the large buffers from imposing excess delay on latency-sensitive, small flows. Traffic from different flows to the same per-rack output queue share bandwidth according to TCP congestion control. The networks *can*, of course, become congested when applications send a large amount of data. This congestion, however, is not unique to c-Through.

To confirm that our per-flow buffering avoids HoL blocking and does not unfairly in-

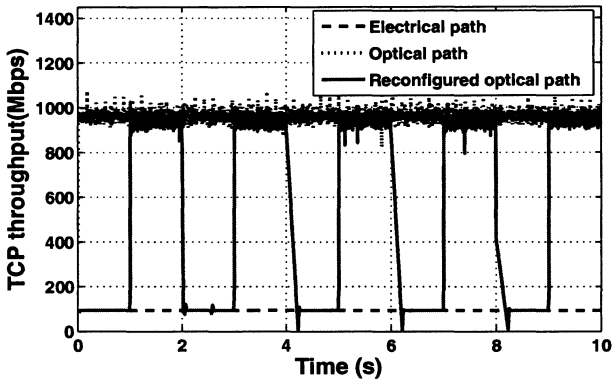


Figure 3.7 : TCP throughput on a reconfiguring path.

crease latency, we send both TCP and ICMP probes together with an aggressive TCP flow to servers in the same destination rack. The probe flows sends 1 probe every second and the heavy TCP flow sends 800MB of data as fast as possible, saturating the 100MB socket buffer and transiently filling the relatively small buffers on the Ethernet switch (until the optical network provides it with more bandwidth in response to its increased demand). Both TCP and ICMP probes observe a 0.15 to 4–5ms increase in RTT due to the congestion at the switch, but do *not* experience the huge delay increases it would observe were it queued behind hundreds of megabytes of data.[5]

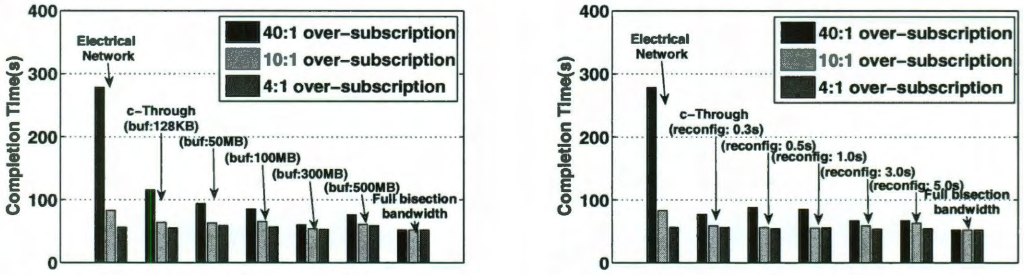
3.7 Applications on c-Through

The benefits from using HyPaC depends on the application communication requirements. The traffic pattern, its throughput requirements, and the frequency of synchronization all affect the resulting gain. We evaluated three benchmark applications, summarized in Table 3.3. We chose these applications to represent three types of cluster activity: bulk transfer (VM migration), loosely-synchronized computation (Map Reduce), and tightly-synchronized computation (MPI/HPC). To understand how applications perform when op-

[5]This design still could increase latency of application control messages *within a flow* if the application sends data and control messages over the same socket.

App	Traffic pattern	Synchronization
VM Migration	one-to-many	NONE
MapReduce	all-to-all	Loose
MPI FFT	all-to-all	Global barrier

Table 3.3 : Benchmark applications



(a) Varying socket buffer size (reconfigure: 1s)

(b) Varying reconfiguration interval (buffer size: 100MB)

Figure 3.8 : Virtual machine migration performance

tical circuits are combined with an electrical network at different speeds, we emulated 40:1, 10:1, and 4:1 over subscription ratios by setting the cross-rack packet-switched bandwidth to 100Mbps, 400Mbps and 1Gbps. Although in practice a data center may run mixed applications, we run each application separately for ease of understanding the applicability of c-Through to different application patterns.

3.7.1 Case study 1: VM Migration

Virtual machine (VM) migration—in which a live VM (including a running operating system) moves from one physical host to another—is a management task in many data centers. The core of migration involves sending the virtual machine memory image (usually compressed) from one host to another. This application represents a point-to-point bulk data

transfer and does not require synchronization with machines other than the sender and receiver. Because the VM memory image may be large (multiple GB) and the migration needs to be done rapidly, VM migration requires high bandwidth. Intuitively, we expect this application to be an ideal case for c-Through: the high bandwidth demand will make good use of the optical network, the pairwise concentrated traffic (point-to-point) is amenable to circuit-switching, and the lack of synchronization means that accelerating part of the traffic should improve the overall performance.

We deployed the KVM [wwwg] VMM on our testbed to study the performance of virtual machine migration with c-Through. Our experiment emulates a management task in which the data center manager wants to shut down an entire rack; prior to shutdown, all the VMs running on that rack need to be migrated to other racks. In our experiments, we migrate all the VMs evenly to servers in other racks based on the VM ID. The starting state has eight VMs per physical node; each VM is configured with 1GB of RAM. Since we have four servers per rack, we need to migrate 32 VMs in total.[6]

Figure 3.8 shows the average completion time of the task, averaged across 3 runs, using c-Through, the bottlenecked electrical network, and a full bisection bandwidth network. Results are further presented as a function of the TCP send buffer size (Figure 3.8(a)) and the reconfiguration interval (Figure 3.8(b)). The three bars correspond to 40:1, 10:1, and 4:1 over subscription ratios (flat for the case of full bisection bandwidth).

We begin by examining the results when the electrical network is 40:1 over-subscribed. In Figure 3.8(a) we fix the reconfiguration interval to be 1s and vary the TCP socket buffer sizes on the servers. Using only the bottlenecked electrical network, the migration takes 280 seconds. Even with default 128KB socket buffers, c-Through accelerates the task to 120 seconds. With larger socket buffers, c-Through improves performance even more. For example, with 300MB TCP socket buffers, the job completes within 60 seconds—

[6]Although each VM is configured with 1GB RAM, the virtual machine may not be using all the RAM at one time. Thus, before migration, the VMM will compress the memory image, reducing the amount of data it needs to send.

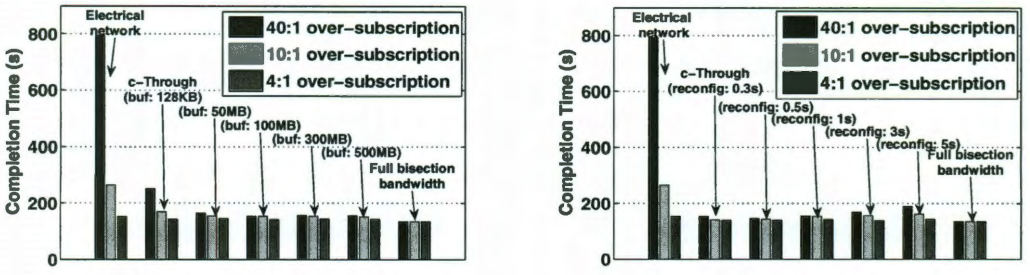
very close to the 52 second completion time with a full bisection bandwidth network. The reasons for this near optimal performance are twofold: First, during VM migration, each source host will tend to have some traffic destined for each rack. As a result, an optical path, once configured, can be fully utilized. Second, the performance of the full bisection bandwidth network is closely determined by the time to initiate the task and compress the VM images, operations that are CPU-limited and not bandwidth-limited. As a result, the bandwidth provided by the optical path, while not always full bisection, is nonetheless sufficient to accelerate performance.

Figure 3.8(b) shows the effect of the reconfiguration interval on VM migration performance. In this figure, we fix the socket buffer size to 100MB, as a reasonable choice from the previous results. Since VM migration is a point-to-point bulk transfer application, it does not require fine-grained optical path reconfiguration. c-Through achieves good VM migration performance even with a 5 second optical reconfiguration interval.

As expected, when the electrical network is less oversubscribed, the performance gap among the three network designs shrinks. That is, when the bandwidth of electrical network is higher, the benefit of adding more bandwidth is relatively smaller. More importantly, however, our results clearly demonstrate that even *augmenting a slow electrical network with c-Through could achieve near optimal performance for VM migration/bulk data transfer applications.*

3.7.2 Case study 2: MapReduce

MapReduce [DG04b] is a widely-used parallel computation approach in today's data centers. Many production data centers, such as Amazon EC2 and Google, support MapReduce-style applications. In MapReduce applications, jobs are split into Map tasks and Reduce tasks. During the Map phase, the master node assigns the job to different mappers. Each mapper reads its input from a data source (often the local disk) as key-value pairs and produces one or more intermediate values with an output key. All intermediate values for a given output key are transfered over the network to a reducer. The reducer sorts these in-



(a) Varying socket buffer size (reconfigure: 1s)

(b) Varying reconfigure interval (buffer size: 100MB)

Figure 3.9 : The performance of Hadoop sort

termediate values before processing them; as a result, it cannot start until all mappers have sent it their intermediate values. At the end of the Reduce task, the final output values are written to stable storage, usually a distributed file system, such as the Hadoop filesystem (HDFS).

In the context of c-Through, MapReduce has two important features: all-to-all traffic and coarse-grained synchronization. The mappers must often shuffle a large amount of data to the reducers. As a result, MapReduce applications may not have traffic that is as concentrated as that in a bulk data transfer such as VM migration.[7] Second, MapReduce applications have only a single system-wide synchronization point between the Map and Reduce phase.

To explore the effect of c-Through on MapReduce-style applications, we deployed Hadoop (an open source version of MapReduce) on our testbed. The first Hadoop application that we ran was a distributed *sort*. A feature of Hadoop *sort* is that the intermediate data generated by the mappers and the final output generated by the reducers are at least as large as the input data set. Therefore, Hadoop's sort requires high inter-rack network bandwidth. Our sort uses 10GB of random data as its input set, and we vary the c-Through

[7]With one important exception: During the output phase, the final values are written to a distributed filesystem, often in a rack-aware manner.

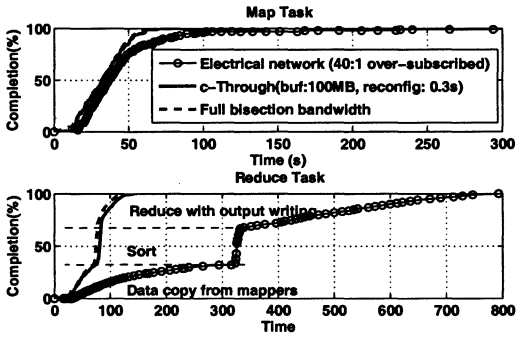


Figure 3.10 : The completion of Hadoop sort tasks

parameters as before.

Figure 3.9 shows the performance of Hadoop sort on c-Through with different buffer sizes, reconfiguration intervals. Figure 3.9(a) shows the effect of the TCP send buffer size on job completion time (as before we fix the optical network reconfiguration time to 1s). Enlarging the TCP socket buffers has very little impact on performance when the buffer size is larger than 50MB. The reason is that the block size of HDFS is 64MB, which also forms its transmission unit. Consequently, Hadoop does not expose enough traffic to the kernel when socket buffers are set to large values.

In Figure 3.9(b), we fix the TCP send buffer size to 100MB and vary the optical path reconfiguration interval. The key result from this experiment is that more frequent optical path reconfiguration can improve Hadoop's sort performance (this should be expected, since faster reconfiguration allows for more frequent draining of the slowly filling buffers).

Coupling c-Through with a slow (40:1 over-subscribed) electrical network provides near-optimal performance for Hadoop sort. To understand the source of this performance improvement, we plot the benchmark's execution timeline in Figure 3.10. The top portion shows the Map phase and indicates the completion of the map tasks as a function of time. The solid line shows Hadoop's performance on the bottlenecked electrical network with 40:1 over subscription ratio; note how the curve has a long tail. This tail comes from a few mappers that are reading from a non-local data source and are hitting the electrical

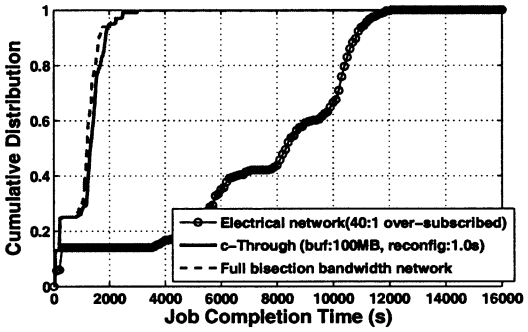


Figure 3.11 : The completion of Hadoop Gridmix tasks

network's bottleneck capacity. By adding optical paths, c-Through eliminates the long tail, significantly reducing the completion time of the map phase (seen when the c-Through curve reaches 100%). This reduction, in turn, improves overall Hadoop sort performance because the reduce tasks cannot start processing until all of the map tasks have completed.

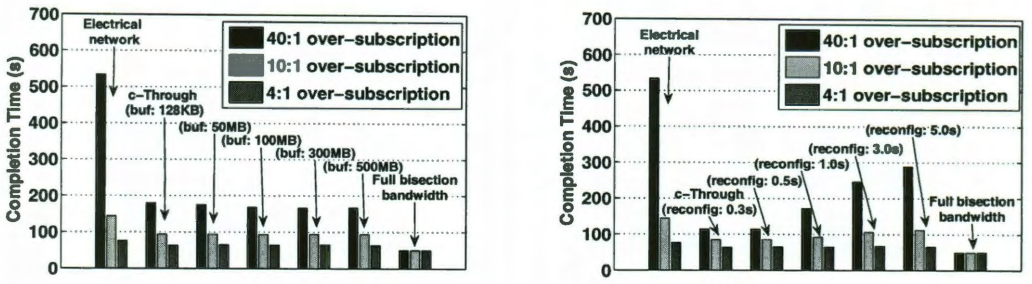
The bottom portion of Figure 3.10 shows the execution timeline of the Reduce phase, which has three steps: copying the intermediate values from the mappers and grouping by key, sorting, and the execution of the reduce function. (The execution of the reduce function also includes writing the output back to HDFS.) c-Through can significantly speed up the data copying and output writing. During the data copying step, data is shuffled among all of the mappers and reducers. Since the use of key space and the value sizes are uniform, there is an equivalent traffic pattern across all racks. Nevertheless, our results shows that c-Through can still accelerate this data shuffling step. Since reducers can start pulling data from mappers as soon as it is available, intermediate data is shuffled among mappers and reducers while the remaining mappers are still running. There is not a blocking serialization point in the shuffle phase because the map tasks and reduce tasks are running in parallel. This property makes it suitable for batch transfer over optical paths.

As reduce tasks complete, the final output is written to HDFS, which maintains (by default) three replicas of each data block. After an invocation of reduce, HDFS will write one copy of the data locally and send two additional copies of the data to other servers. This

replication generates significant network traffic and saturates the electrical network, which is clearly visible in the bottom figure by the long tail of the output writing step. Again, this traffic does not exhibit significant skew, but nevertheless, c-Through can accelerate this bulk data transfer step significantly.

Finally, we summarize MapReduce's performance on the three network designs. When Hadoop sort runs on a full bisection bandwidth network, it can still be bottlenecked by intensive disk I/O operations. Consequently, as before, we observe that even using reconfigurable optical paths with a slow electrical network can still provide close to optimal performance to data-intensive, MapReduce-style applications—despite relatively uniform traffic patterns. Similarly, when the electrical network becomes faster, the performance gap among bottlenecked electrical network, c-Through and full bisection bandwidth network is smaller.

Gridmix: To understand how c-Through works for more realistic applications with complicated traffic patterns, we study the performance of the Hadoop Gridmix benchmark on c-Through. Gridmix [wwwe] mimics the MapReduce workload of production data centers. The workload is simulated by generating random data and submitting MapReduce tasks based on the data access patterns observed in real user jobs. Gridmix simulates many kinds of tasks with various sizes, such as web data scanning, key value queries and streaming sort. We generate a 200GB uncompressed data set and a 50GB compressed data set for our experiments. Each experiment launches 100 tasks running on these data sets. We run the same Gridmix experiment 3 times for each network architecture. Figure 3.11 shows the cumulative distribution of the completion time of Gridmix tasks on the bottlenecked electrical network, full bisection bandwidth network and c-Through network respectively. In our experiment, c-Through uses 100MB socket buffer size and 1 second optical reconfiguration time. The over-subscription ratio of the electrical network is 40:1. The results show that even when the electrical network is highly over-subscribed, the completion time of Gridmix jobs on c-Through network is very close to that on full bisection bandwidth



(a) Varying socket buffer size (reconfigure:1s)

(b) Varying reconfigure interval (buffer size: 100MB)

Figure 3.12 : The performance of MPI FFT

network.

3.7.3 Case study 3: MPI Fast Fourier Transform

Fast Fourier Transforms (FFTs) are frequently used in scientific computing; efficient parallel FFT computation is important for many large scale applications such as weather prediction and Earth simulation. Parallel FFT algorithms use matrix transpose for FFT transformation. To perform a matrix transpose using MPI, a master node divides the input matrix into sub-matrices based on the first dimension rows. Each sub-matrix is assigned to one worker. Matrix transpose requires each worker to exchange intermediate results with all other workers. Parallel FFT on a large matrix therefore requires data intensive all-to-all communication and periodic global synchronization among all of the workers. We expected this style of application to be challenging to a hybrid network because of the poor traffic concentration and strict synchronization among servers. Surprisingly, c-Through substantially improved performance.

We studied the performance of parallel FFT on our testbed using FFTW [wwwc]. FFTW is a C library for computing discrete Fourier transform in one or more dimensions. It provides both single node and MPI-based implementations to compute FFT with real or complex data matrices. We ran MPI FFTW on 15 nodes of our testbed to compute the

Fourier transform of a complex matrix with 256M elements. The resulting matrix occupies 4GB. Figure 3.12 reports the MPI FFT completion time with different network settings.

We follow the same method as with previous experiments to study the effect of socket buffer size and optical reconfiguration interval on application performance. Because MPI FFT exchanges small blocks (20MB) of the matrix among servers, larger buffers do not substantially improve performance. The benefits of c-Through for MPI FFT depend more on rapid optical path reconfiguration. As shown in Figure 3.12, with 100MB socket buffers and a 0.3 second optical reconfiguration interval, the job took twice as long using c-Through with a 40:1 oversubscribed electrical network as it did on the full bisection bandwidth network. When the optical path is reconfigured slowly (e.g. 5 seconds), the performance of MPI FFT is much worse than when the reconfiguration interval is 0.3 seconds. The frequent global barrier synchronization, combined with the uniform traffic matrix, make it impossible to saturate the optical network for the entire period. *To maximize performance, applications must ensure there is enough data available in the buffers to saturate the optical link when it becomes available.* Reconfiguration periods should not be large without reason: The period should be chosen just long enough to ensure efficiency (the network experiences downtime during the reconfiguration), but small enough so that links do not fully drain due to lack of data to the chosen destination rack.

3.8 Discussion

3.8.1 Applicability of the HyPaC Architecture

The case studies provide several insights into the conditions under which a HyPaC network can or cannot provide benefits.

Traffic concentration: Bulk transfers can be accelerated by high capacity optical circuits. c-Through buffers at sources to create bulk transfer opportunities, but applications may not be able to make full use of such a feature if they internally operate on small sized data units. An example of this comes from our experience with Hadoop: In many

cases, Hadoop's default configuration limited the nodes' ability to generate and buffer large amounts of data, even when TCP provided sufficient buffer space. Fortunately, the performance tuning was straightforward—and similar to the techniques needed to achieve high throughput on high bandwidth-delay networks.

Even applications that may seem on the surface to have uniform traffic matrices may experience transiently concentrated traffic, such as the data output phase in Hadoop. By diverting even just this one phase of the transfer over the optical links, all applications running on the cluster—even those with uniform traffic—can benefit from the reduced load on the oversubscribed electrical network. Second, even uniform applications such as the sort phase of Hadoop may experience traffic concentration on shorter timescales due to statistical variation in their traffic. If those timescales are long enough, the hybrid network can exploit the transient imbalance to accelerate the transfer.

Synchronization: HyPaC networks are more suited for applications with loose or no synchronization. The pairwise connections and reconfiguration interval impose a minimum time to contact all racks of interest called the *circuit visit delay.* If the time between application synchronization events is substantially smaller than the circuit visit delay, the benefits of a HyPaC network decrease rapidly. There are two ways to reduce the circuit visit delay: Cluster the application traffic so that it must visit fewer racks, or move to faster optical switching technologies; until these technologies become available, the HyPaC architecture may not be an appropriate choice for tightly synchronized applications that require all-to-all communication. We discuss the scheduling algorithms that are needed to support explicit traffic dependency over optical circuits in Chapter 4.

Latency sensitivity: All the applications we have studied so far are not sensitive to the latency of particular messages. Some data center applications, such as Dynamo [DHJ+07], do not operate on bulk data. Instead, they need to handle a large number of small queries. These applications are sensitive to the latency of each query message. For such applications, a HyPaC network can improve query throughput and relieve congestion in the electrical network, mostly because these applications also perform concentrated bulk transfers

during reconfiguration and failover. However, it does not reduce the non-congested query latency.

3.8.2 Making Applications Optics Aware

Our current design makes the optical paths transparent to applications. Applications might take better advantage of the HyPaC architecture by *informing* the optical manager of their bandwidth needs, or by *adapting* their traffic demands to the availability of the optical network. We view both of these questions as interesting avenues for future exploration. The applications we have studied so far have largely stationary traffic demands that would benefit little from telegraphing their intent to the manager. Other applications, however, may generate more bursty traffic that could benefit from advance scheduling. The second approach, allowing applications to actively adapt to the reconfiguring bandwidth, could potentially leverage optical paths in much better ways.

For example, applications could buffer more data on their own, or generate data in a more bursty fashion based upon the availability of the network. Several datacenter applications are already topology-aware, and it may be possible to make such applications (e.g., Hadoop) adapt to the changing topology just by modifying the scheduling algorithms. Finally, the optical component manager might be integrated into a cluster-wide job/physical resource manager that controls longer-term, high level job placement, to improve traffic concentration as discussed below. In Chapter 6, we discuss a specific idea of customized data aggregation scheme to accomplish data shuffling more efficiently over optical circuits.

3.8.3 Scaling

There are three scaling challenges to realize a large hybrid network. First, the size of the data center may exceed the port capacity of today's optical switches, requiring a more sophisticated design than the one considered so far. Second, the measurement and matching computation overhead scales with the number of racks. Third, if racks talk to more and more other racks, the *circuit visit delay*, i.e. the amount of time it takes to connect one rack

to a particular other rack, might increase. To make the discussion concrete, we consider a large data center with 1000 racks (a total of 40,000 servers).

Optical network construction: Given a 1000-rack data center, c-Through would require a 1000-port optical circuit switch. Research prototype MEMS optical switches already scale to a few thousand input and output ports [KNK+03], but they are not available commercially; today's largest MEMS switches has 320 input and output ports [wwwb]. In the future, however, this approach could provide a very simple and flexible way to add optical paths into large data centers.

Without such dense switches, one choice is to divide the datacenter into zones of 320 racks (12800 servers). As we discuss below, both anecdotes and recent datacenter utilization data from Google suggest that the vast majority of jobs run on clusters smaller than this. However, this answer is both intellectually dissatisfying and is based only on *today's* use patterns. An alternate approach is to chain multiple MEMS switches using the same fat-tree or butterfly topologies used in recent research to scale electrical networks, at the cost of both increased cost and the requirement for fast, coordinated circuit switching. Fortunately, constructing a rack-to-rack circuit switched optical fat tree is much simpler than a node-to-node packet switched fat tree (it operates at roughly $\frac{1}{40}$ the scale). Eleven 320x320 optical MEMS switches could cover a 1000-rack data center, at a cost three times higher per node than simply dividing the network into zones.

Optical network management: The VLAN based traffic control can remain unchanged when c-Through is used in large data centers since it operates on a per-switch or per-server basis. If we use a single switch or a fat-tree to construct the optical network, Edmonds' algorithm can still be used to compute the optical path configuration. Edmonds' algorithm is very efficient: For a 1000-rack data center, the computation time to generate a new configuration/schedule is only 640ms on a Xeon 3.2GHz processor, which is sufficiently fast for a large class of data center applications [WAK+09]. If the optical network topology is a hierarchy or a ring, the configuration is not a perfect matching anymore; nevertheless, it is still a max weighted matching. Several algorithms can compute the max-

imum matching efficiently [Gal86], and several approximation algorithms [Gal86, Mck99] can compute near-optimal solutions even faster.

Scaling also increases the demands on collecting and reporting traffic measurements, as well as disseminating reconfiguration notices. If optical paths are reconfigured every second and all of the rack queues have a large amount of data buffered, each server needs to report its traffic demands (4 bytes each) for each of the 1000 possible destination racks. Thus, each server will send 4KB of traffic statistics data to the optical manager. Across the whole data center, the optical manager will receive 160MB of traffic measurement data (40 servers per rack) every second. We believe that our existing heuristic for discarding under-occupied buffer slots will be sufficient to handle this aspect of scaling (Section 3.5). For example, based on our MapReduce experiment trace, when servers only report queue sizes larger than 1MB, the traffic measurement data is only 22% of the worst case volume. As discussed below, we believe that as the network scales, more and more of the rack-to-rack paths are likely to be unused.

Finally, a number of standard techniques, such as hierarchical aggregation trees and MAC-layer multicast, can help alleviate the messaging overhead. Multiple message relay nodes can be organized into a logically hierarchical overlay. Each leaf node collects traffic measurement data from a subset of servers, aggregating the data through the hierarchy to the root that computes the optical configuration. Notifications of configuration changes can also be disseminated efficiently through the hierarchy to the servers. Intra-rack MAC layer multicast could be used to further improve efficiency.

Circuit visit delay: The circuit visit delay depends on the number of destination racks to which servers spread traffic. In a 1000-rack data center, the worst case scenario is an application that simultaneously sends large amounts of traffic to destinations in all 1000 racks. The last rack will have to wait for 999 reconfiguration periods before it can be provisioned with an optical path.

The problems of long *circuit visit delay* are twofold. First, the amount of data that must be queued could grow quite large—up to a full circuit visit delay times the server's link

capacity. At 1 Gb/s, 999 seconds worth of data (128 GB) would substantially exceed the memory available in most servers. Worse, this memory is pinned in kernel. During the wait for reconfiguration, the application's traffic will be routed over the electrical network, causing congestion for other, possibly latency-sensitive, traffic.[8] For this worst case scenario, data center designers may require other solutions, such as a full bisection bandwidth electrical network.

In practice, however, many factors can help keep the circuit visit delay low. First, although some applications (e.g. MapReduce, MPI) require all-to-all communication, the application may only use a smaller set of racks during any particular phase of its execution. Second, in a large data center, a single application usually does not use the entire data center. In many cases, a large data center is separated into many subnets. Different applications are allocated into different subnets. Although we do not have exact numbers about the job sizes in production data centers, recently released workload traces from a Google production data center show that their largest job uses 12% and the medium size job only uses 0.8% of cores in the data center [wwwd].[9] Therefore, even in a large data center, there are likely to exist smaller cliques of communicating nodes. In fact, Microsoft researchers have also observed this to be true in their large production data center [KPB09]. A second path to scaling, then, is to ensure that the job placement algorithms assign these applications to as few racks as possible. This is actually a simplification of the job placement algorithms required today, which must not only group at the rack level, but at the remaining levels of the network hierarchy. In Chapter 6, we discuss a partial aggregation mechanism that can accomplish all-to-all data shuffling over optical circuits in $log(N)$ (instead of N) rounds.

[8] We have deliberately avoided a design in which some traffic is delayed until the optical network becomes available, feeling that such an approach makes little sense without tight application integration to appropriately classify the traffic.

[9] The data do not specify the exact size of the datacenter.

3.9 Summary

Building full bisection bandwidth networks using packet switches imposes significant complexity. It may also aim to provision more than today's and future's applications require. We explore the use of optical circuit switching technology to provide high bandwidth to data center applications at low network complexity. We present a HyPaC architecture that integrates optical circuits into today's data centers, and demonstrate the feasibility of such a network by building a prototype system called c-Through.

By studying several modern datacenter applications, we assess the expected gain from integrating optical circuits in target data center scenarios. Our results suggest that a HyPaC network offers the potential to significantly speed many applications in today's datacenters, even when the applications may not intuitively seem to be promising candidates for acceleration through circuits. While there remain many significant questions about—and options for—the design of future datacenter networks, we believe the HyPaC architecture represents a credible alternative that should be considered along-side more conventional packet switched technologies.

Chapter 4

Managing Optical Circuits in Heterogeneous Data Centers

4.1 Motivation and Solution Outline

In the previous chapter, we have discussed the feasibility and basic design of HyPaC network architecture to provide high bandwidth connectivity in data centers. Other than our c-Through system, UCSD researchers also built a hybrid network called Helios [FPR+10] which shares the similar concept with ours. Both create a *hybrid* network that combines the best properties of electrical packet switches and high-bandwidth optical circuit switches, the latter reconfigured at millisecond timescales using MEMS optical switches.

These efforts demonstrated the promise of deploying hybrid networks in commodity Ethernet environments. A number of important cloud applications, MapReduce, virtual machine migration, large data transfers, demonstrated significant performance improvements while running on such a hybrid network with the potential for much lower cost, deployment complexity, and energy consumption.

Unfortunately, both studies ignored an important property of modern cloud datacenters: their fundamental heterogeneity and multi-tenancy. We have since encountered a number of challenges, some quite unexpected, in building these networks and using them for larger-scale, mixed workloads. In this chapter, we discuss both the promise of hybrid networks and the importance of addressing the key challenges that can prevent their wider adoption.

Several hardware/firmware engineering challenges arise in incorporating switched optical circuits into the modern datacenter. Most notably, the link setup and switching time of optical components are far from their theoretical limits—perhaps because they were previously used primarily for slowly switching (tens of seconds or longer) telecom applications.

The software challenges are more complex, with the most thorny arising from the tem-

poral and spatial heterogeneity in datacenter traffic: flow duration, correlations in demand and interference between applications, priority across flows within and across applications. Addressing these software challenges requires near real-time analysis of application requirements to support dynamic switch scheduling decisions. Ideally such measurement and analysis should take place on aggressive timescales of milliseconds or tens of milliseconds across networks of tens of thousands of servers.

The original Helios and c-Through designs focused on maximizing total bisection bandwidth, treating flows as interchangeable and undistinguished from each other—effectively ignoring the diversity described above. Although a reasonable starting point, these assumptions often produce circuit scheduling behavior and forwarding decisions that lead to unnecessary circuit flapping and low circuit utilization. This chapter revisits several of these assumptions to articulate design goals for practical hybrid datacenter interconnects; specifically, the interconnect should 1) tolerate greedy and ill-behaved flows that try to occupy circuits but not use them, 2) tolerate inaccurate information about application demands and changes, 3) support flows that are inter-dependent and correlated, and 4) support flexible circuit sharing policies with flow and application differentiation.

Based on these challenges, our design goals, and our year-long experiences using hybrid networks, we propose a meta-solution to these challenges: that the control framework for a hybrid datacenter interconnect should allow flexible, fine-grained, and responsive control. The framework collects traffic statistics and network status information from various sources, performs data analysis to understand traffic demand with application semantics, and configures the network with user-defined objectives about overall performance and sharing metrics. It should do so without imposing overhead or delays that would prevent reacting on millisecond timescales.

We therefore introduce an *observe-analyze-act* framework for optical circuit configuration. We explore the circuit scheduling algorithms that can incorporate traffic dependencies at application level. We have begun prototyping an OpenFlow [MAB+08] implementation of this framework; a modular topology manager controls the placement of individual flows

onto optical circuits or electrical packet switches. Our experience using this framework suggests that the fine granularity supported by per-flow placement supports the implementation and testing of advanced scheduling algorithms capable of capturing application and datacenter demands.

4.2 Overly Restrictive Design Assumptions of c-Through and Helios

Both c-Through and Helios estimate the network traffic demand, identify racks that have long-lived, stable aggregated demand between them, and establish circuits between them to amortize the high cost of switching circuits. The two systems use different techniques to estimate traffic demand, but use the same greedy approach to schedule optical circuits to maximize the amount of data sent over optical circuits. Their centralized controllers dynamically configure the circuit and packet switches based upon the greedy scheduler's output every reconfiguration interval. Neither system incorporates history or state into their scheduling. These current proposals for hybrid networks make five overly restrictive assumptions about the traffic on the hybrid network:

Flows are independent and uncorrelated: Assigning circuits between two racks will not affect the bandwidth desired by other flows in the system.
Reality: Many flows depend on the progress of other flows, such as those that relay traffic through a node.

All flows have same priority: Helios and c-Through schedule optical circuits without taking into account any application or flow level traffic prioritization.
Reality: Datacenter traffic has a rich mix of priorities and fair sharing desires within and between applications.

Flows will not underutilize the circuits: Once flows are assigned to circuits, they will continue to use the same (or more) bandwidth as their predicted demand indicated until the

control process reassigns that circuit elsewhere (which might not happen for hundreds of milliseconds or more).

Reality: Perfectly predicting future traffic demand is not as easy as one might hope.

Randomly hashing flows to optical circuits is effective: Given a set of N 10Gbps circuits between two racks, we can create a single *superlink* of $10N$ Gbps capacity using random hashing, in which flows are randomly hashed across the set of available circuits.

Reality: Random hashing cannot support flexible distribution of circuit bandwidth that maximizes application performance.

All flows that can use a circuit should use that circuit: There is no cost to using an optical circuit when it is available, or to switching flows between the electrical and optical circuits. The Helios design made this assumption because of switch software limitations—traffic could not be scheduled on a per-flow basis to the electrical or optical network. The c-Through design made the same assumption to keep its optical/electrical selection tables small using only per-destination entries.

Reality: We have found several cases where keeping some lower-bandwidth, latency-sensitive flows on the packet switch reduces latency and variance for some applications.

4.3 Challenges to integrating optics into real datacenters

This section demonstrates experimentally specific application traffic patterns that expose weaknesses in both Helios and c-Through. We work with UCSD researchers and integrate c-Through and Helios systems onto an unified hybrid network testbed. Figure 4.1 shows our prototype hybrid testbed with both electrical and optical circuit switches. The testbed has 24 servers organized into 4 racks, with 6 servers in each rack. The servers are configured with 10Gbps NICs. Servers racks are connected with two switches, one 64-port Glimmerglass optical circuit switch with 10Gbps optical interface and one 10Gbps Ethernet switch. The optical switch can provide 5 optical up links for each server rack. We

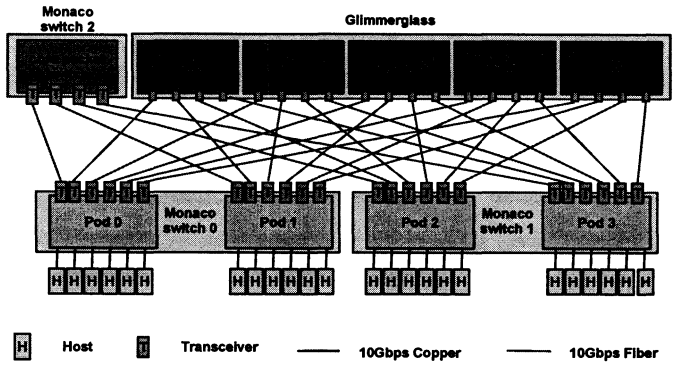


Figure 4.1 : Hybrid testbed with 24 servers, 5 circuit switches and 1 core packet switch.

implement an unified optical circuit controller that can collect traffic demand from both servers (as in c-Through) and switches (as in Helios) and compute circuit configuration using Edmonds' algorithm.

4.3.1 Effect of bursty flows

Both c-Through and Helios use the greedy Edmonds' matching algorithm [Edm65] to assign circuits between racks, based on an instantaneous snapshot of the traffic demand. Unfortunately, scheduling using an instantaneous demand estimate can find a locally maximal circuit configuration in which some circuits are under-utilized, while there exists another configuration with better overall circuit utilization.

We demonstrate this problem using a topology of three racks (rack 0, 1 and 2), with two hosts each. Each rack switch has one 5Gbps uplink to the core packet switch and one 10Gbps uplink to the optical circuit switch. In this topology, only two racks can be connected optically at any given time. For the duration of the experiment, one host in Rack 1 sends data to a host in Rack 2 over a long-lived TCP connection (the "foreground flow"). The second host in Rack 1 sends data to a host in Rack 0 following a bursty ON-OFF pattern; we vary the burst ON-duration with the OFF-duration set to 2 second to observe the circuit scheduling decisions made by the Helios and c-Through circuit schedulers.

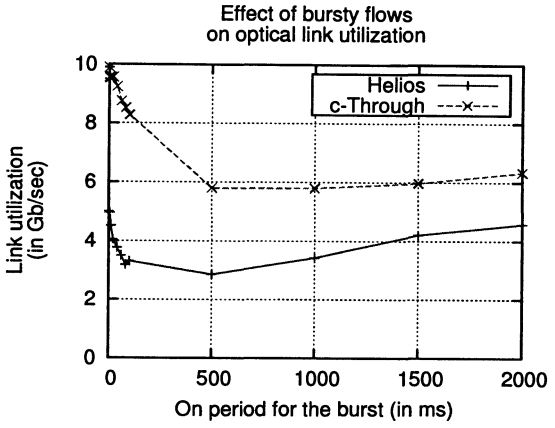


Figure 4.2 : The utilization of a circuit in the presence of a bursty flow. Note that by attempting to schedule the bursty flow on the circuit, bandwidth is reduced for the long-lived foreground flow.

Figure 4.2 shows the average utilization of the optical link. In both designs, as the duration of the traffic bursts increases, the utilization of the link initially decreases and then increases as the bursts grow longer than 500ms. With short bursts, the circuit is assigned to the bursty flow between Rack 1 and Rack 0, but after assignment, this flow goes quiescent, under-utilizing the optical capacity. In the next control cycle, the circuit is assigned back to the long-lived foreground flow. The control cycle is hundreds of milliseconds; bursts shorter than the control loop will reduce utilization for part of the control loop cycle. Longer bursts use the optical circuit for a longer fraction of the time it is assigned, improving overall optical utilization. Notably, the optical link capacity is never saturated by the long flow because of the constant flapping of the circuit between the racks.

4.3.2 Effect of correlated flows

An important component of datacenter traffic has a "shuffle", or all-to-all workload characteristic. For example, MapReduce [DG04a] and Hadoop [had] require large scale sorting and shuffling of data between nodes. In this section, we evaluate the performance of a representative large scale sorting application, TritonSort [RPC+11]. We use TritonSort on

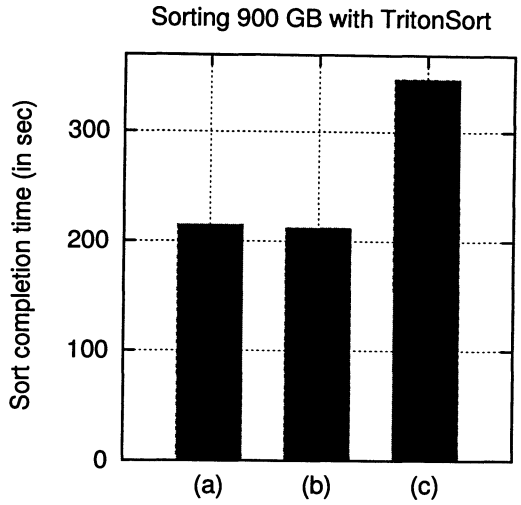


Figure 4.3 : TritonSort performance in the following cases: (a) Fully nonblocking electrical switch (baseline case) (b) Helios/c-Through network with no background flows (c) Helios/c-Through network with one competing background flow

Helios to show how suboptimal circuit scheduling impairs performance. TritonSort represents a balanced system where progress is largely a function of the speed of the slowest flow; therefore, allocating high bandwidth to only few TritonSort flows will not improve overall performance. The key deficiency of the Helios and c-Through schedulers is that they do not take into account these dependencies across application flows.

This experiment measures the completion time of TritonSort running together with a long-lived background TCP flow that competes for circuit capacity. We use four racks, each connected with a 5Gbps uplink to the core packet switch and a 10Gbps uplink to the circuit switch. TritonSort sorts 900GB of data using nine nodes (three hosts in each of racks 0, 2, and 3). Meanwhile, another host in Rack 2 sends traffic to a host in Rack 1, competing with TritonSort for the Rack 2 circuit uplink. With only one circuit uplink, Rack 2 can connect optically to only one other rack at a time; if the scheduler creates a circuit from Rack 2 to Rack 1, the TritonSort flows from Rack 2 are sent over the slower packet switch. This imbalances the system, slowing it down. We measure the effect of this background flow on the completion time for TritonSort.

Figure 4.3 shows the completion time for sorting 900GB of data when the racks are inter-connected by 1) a fully nonblocking electrical switch (the baseline case), 2) Helios/c-Through network with no background flow, and 3) Helios/c-Through network with the background flow. Absent other traffic, Helios matches the performance of a non-blocking switch, but in the presence of competing flows, TritonSort's completion time increases substantially, taking as much as 69% longer.

4.3.3 The Hashing effect on multiple circuits

Datacenter networks increasingly use multi-path topologies to increase reliability and performance. Their switching hardware computes a hash of the packet header's source and destination information to determine which of several equal-cost paths to send each flow on; this mechanism ensures that packets within a flow follow the same path to avoid reordering, which can cause problems with TCP. While hash-based load balancing works well when the number of flows is large and individual flows are relatively small, it can perform worse under several real application workloads. For example, in a workload where most of the flows are mice (small) with only a few elephant flows, hashing uniformly across the entire set of flows can result in the elephant flows receiving too little bandwidth. The following experiment illustrates this potential problems of using random flow hashing to individual circuits.

Four hosts in Rack 0 send data to four hosts in Rack 1. These four flows between two racks are hashed over a circuit path with four individual circuit uplinks. If the hashing is truly uniform, four flows would be hashed over four circuits separately, and each flow get 10Gbps throughput. However, we observe that two of the flows get hashed to the same circuit due to hash collision. As a result two flows receive 10Gbps but the other two flows receive only 5Gbps and hence one of the four circuits is not utilized.

4.3.4 Hardware issues

Other than the application and network level challenges, there are a few hardware hurdles to achieving fast and efficient network reconfiguration in HyPaC networks. The major problem is that current optical components are not designed to support rapid reconfiguration due to their original requirements in the telecom industry, and thus there has been a lack of demand to develop faster optical control planes.

The switching times of current optical switches are still far from their ideal limits. The best switching time is still around 10 to 25 ms for commercially available optical switches. However, free-space MEMS switches should be able to achieve switching times as low as 1ms. Another hardware issue comes from the design of optical transceivers. Currently available transceivers do not re-sync fast enough after a network reconfiguration. The issue is that the time between light hitting the photoreceptor in the transceiver, until the time that an actual data path is established is needlessly too long–as long as several seconds in practice [FFL+11]. The underlying transceiver hardware can theoretically support latencies as low as a few nanoseconds. Beyond the physical limitations, much of the control plane performance limitations are due to the electrical control aspects of the design, including the implementation of the switch software controller.

4.4 Solution Space

This section explores the requirements for datacenter optical circuit controllers and sketches the design of an observe-analyze-act framework that can (help) meet them.

4.4.1 Design requirements

An optical circuit controller should be able to meet four key requirements:

Tolerate inaccurate demand information: It is difficult for a controller to have precise knowledge about all application's traffic demands and performance requirements. Existing systems infer these demands from counters in the network, but as we have shown, these

heuristics can result in flapping circuits and sub-optimal network performance. A good circuit allocation mechanism must be robust to inaccurate measurement of application traffic demands.

Tolerate ill-behaved applications: As we showed in previous sections, bursty datacenter applications can cause unnecessary network reconfiguration. Non-malicious but selfish applications could claim to need more capacity than they really do. All such ill-behaved applications can reduce the performance of a HyPaC network. The circuit controller must therefore be robust to their behavior, ensuring that the network's performance is not affected by them.

Support correlated flows: To achieve good application layer performance, the circuit scheduling module must be able to accommodate flows whose demand and performance depends on the performance of another flow. The underlying framework supporting the scheduler must provide fine-grained control to handle differently traffic that is on the critical path of an application vs. less important flows. It should also provide an avenue for the controller to gather sufficient information to understand the application's dependence upon a flow's performance.

Support flexible sharing policies among applications: Allocating circuits among mixed applications is challenging given the diversity of datacenter applications. Particularly in a multi-tenant cloud environment, applications may have very different traffic patterns and performance requirements. In addition, the management policies in datacenters could assign applications different priorities. To share the limited number of optical circuits among these applications, the circuit allocator must be able to handle the performance interference among applications and support user-defined sharing policies among applications.

To achieve these design requirements, we focus on three design points of the circuit controller: traffic monitoring and analysis component to understand data center traffic with application semantics; intelligent circuit scheduling algorithm that can support correlated traffic and flexible traffic control component to support fine-grain control policies.

4.4.2 An Observe-Analyze-Act framework

The circuit controller must be able to obtain a detailed understanding of application semantics and fine-grain control of flows forwarding policies. We propose a three phase approach for managing HyPaC networks based on the *Observe-Analyze-Act* framework. To get a better understanding of the network dynamics and application heterogeneity in the cloud ecosystem, the HyPaC scheduler should be able to interact with different components and collect information from them. This information collected in this *Observe* phase would include the link utilization from the switches and application status from the cluster job schedulers (e.g., the Hadoop job Tracker), as well as application priorities and QoS requirements. The HyPaC manager then analyzes the aggregation of this information to infer the most suitable configuration for the network. The *Analyze* phase is a key step that helps the network controller understand the application semantics of traffic demand, detect ill-behaved applications, discover the correlated flows and therefore make the optimal configurations to support these applications. Finally, in the *Act* phase, it communicates this configuration to the other components in the system in order for the decision to be acted upon. The *Act* phase requires fine-grain control on the flow forwarding to support flexible configuration decisions and sharing policies. In the following sections, we discuss the key components to realize such a control framework, including circuit configuration algorithm, traffic monitoring and analysis and OpenFlow based control.

4.4.3 Circuit configuration algorithm supporting correlated flows

We first perform an analysis of the algorithmic design of circuit configuration problem. As we have discussed in the previous chapter, the basic circuit configuration problem can be formulated as computing maximum weight perfect matching over the traffic demand matrix to maximize the overall throughput of optical circuits. The traffic demand matrix can be defined by a graph $G = (V, E)$, where each vertex in V represent a rack in the data center and edge $e = \{v_i, v_j\}$ is weighted by the traffic demand cross rack v_i and v_j. We have also known that maximum weight perfect matching problem can be solved

efficiently by existing algorithms, such as Edmonds' matching algorithm. Beyond the basic circuit configuration problem, we look into circuit configuration problem that can support correlated flows.

4.4.3.1 Modeling correlated traffic

We use a simple model to study the dependency of correlated flows. Although there are different dependency structures among data center applications, we assume the all-to-all dependency structure in this study. All-to-all traffic dependency is a common dependency structure appeared in many data center applications, such as sorting, MapReduce and many scientific computing applications. We leave the study of other dependency structures in future work. In the all-to-all dependency structure, there are mutual dependency among all the correlated flows, which means the rate of correlated flows are decided by the rate of the slowest flow. In the context of circuit configuration, it means if we want to accelerate a set of correlated flows, we have to configure optical circuits for all the flows simultaneously. Otherwise, any one flow on the slow path will reduce the traffic rate of other flows even they are routed over high bandwidth optical circuits.

To take into account the correlated traffic in circuit configuration, we introduce *correlated edge groups* into the traffic demand graph to describe these correlated flows. A set of correlated flows can be described as a correlated edge group $EG = \{e_i, e_i \in E\}$ from traffic demand graph $G = (V, E)$. Since the rate of correlated flows could be higher when all the correlated flows are configured to optical circuits, the weight of each edge e_i in the edge group EG is defined as follows:

$$W(e_i) = \begin{cases} W_b(e_i) + \Delta(e_i), & EG \subseteq M \\ W_b(e_i), & otherwise \end{cases} \tag{4.1}$$

Here M is a matching over graph G that represents the set of edges configured with optical circuits, $W_b(e_i)$ is the basic weight of an edge e_i and $\Delta(e_i)$ is the increased weight on edge e_i.

4.4.3.2 Definition of circuit configuration problem with correlated edges

We now formalize the circuit configuration problem with correlated edges as follows.

Input:

- Traffic demand matrix defined by a graph $G = (V, E)$. V is a set of nodes where each node represent one rack; E is a set of edges $E \subseteq V \times V$. An edge $e = v_i \times v_j$ is weighted by the traffic demand across rack v_i and v_j.
- A set of correlated edge groups $S_{EG} = \{EG_1, EG_2, ..., EG_N\}$. EG_i is a correlated edge groups on graph G, where the weight of edges in EG_i are defined as in equation 4.1.

Output:

A perfect matching M in graph G with maximum overall weight. [1]

The difference between this matching problem and classic maximum weight perfect matching problem is that, in classic matching problem, the weight of edges are defined statically in the input graph; while in our matching problem, the weight of edges in correlated edge groups could change depending on the condition if the corresponding edge groups are selected as a subset of final matching.

4.4.3.3 The complexity analysis

For a graph $G = (V, E)$ with correlated edge groups, we define a base graph $Base(G) = (V, E), \{W(e) = W_b(e), \forall e \in E\}$. The base matching M' is the maximum weight perfect matching computed according to the base graph $Base(G)$. We analyze the complexity of this circuit configuration problem by considering the incremental improvement over the base matching. The intuition is, we consider all the correlated edge groups as potential improvement from the base matching.

[1] A matching in graph G is a set of edges without common vertices. A perfect matching is a matching which matches all vertices of the graph.

We start from the simplest case when there is only one edge group. In this case, we can decide "accept" or "reject" the correlated edge group based on if accepting the edge group will result in higher overall weight for the final matching. We define a benefit function given an edge group EG and graph G as follows.

$benefit(G, EG) = Weight(EG + Edmonds(G - EG)) - Weight(Edmonds(G))$

In the base case, the matching is computed by Edmonds' algorithm over graph G. If the edge group is accepted, the final matching can be computed using Edmonds' algorithm over the remaining graph $G - EG$ which excludes edge group EG from graph G. So the benefit of edge group EG is defined by the weight difference between these two cases.

When there are multiple edge groups in S_{EG}, some of the edge groups might conflict with others when they share common vertices. To understand the complexity of this problem, we consider two scenarios separately.

Case1: there is no conflict among edge groups

We show that in the case when there is no conflict among edge groups, we can find the optimal circuit configuration with maximum weight using a greedy algorithm.

Graph MG = G;
for all EG_i in S_{EG} **do**
  if EG_i is not a matching **then**
    continue;
  end if
  if $benefit(MG, EG_i) > 0$ **then**
    Put EG_i as part of final matching;
    $MG = MG - EG_i$;
  end if
end for

Lemma 1. *If there is no conflict among all the edge groups in S_{EG}, we can find the optimal circuit configuration with maximum weight by the greedy algorithm.*

Proof.

We prove the lemma by arguing that when there is no conflict among all the edge groups, the decision of accepting or rejecting an edge group into the final matching can be made by only considering the weight improvement of this edge group.

For an edge group EG_i, if EG_i is not a matching itself, it is not possible to accept EG_i into the final matching. Given a graph MG, and an edge group EG_i which is a matching, there are two potential choices in the circuit configuration result. In option 1, we ignore the weight changes of edge group EG_i and just compute the maximum weight matching using Edmonds' algorithm. The resulting matching has the maximum weight without considering edge group EG_i. In option 2, we accept EG_i as part of the final matching, and compute the maximum weight matching over the remaining part of the graph (denoted as $MG - EG_i$) using Edmonds' algorithm.

We can decide if we accept EG_i into the final matching by comparing the weight of the above two options. If option 2 has higher overall weight, we can accept EG_i into the final matching no matter what. This decision will result in a final matching with higher overall weight because (1) we have known that this option has higher weight than any other options without considering EG_i (option1); (2) since there is no conflict among all the edge groups, whether or not accepting remaining edge groups will not impact the decision on EG_i.

Therefore, we can achieve maximum weight matching by independently accepting all the edges groups that can improve the overall weight of the matching.

Case2: there are conflicts among edge groups

The problem is more complicated when there are conflicts among edge groups in S_{EG}. We show that the problem is NP-hard because it is equivalent to maximum independent set problem over a conflict graph for all the edge groups.

Lemma 2. *When there are conflicts among edge groups in S_{EG}, finding the optimal circuit configuration with maximum weight is NP-hard.*

Proof.

When there are conflicts among edge groups in S_{EG}, finding the optimal circuit configuration with maximum weight is to find a set of coexisting edge groups with maximum weight improvement over the base matching. To solve this problem, we can construct a conflict graph $FG = (S_{EG}, CE)$ for all the edge groups. The conflict graph is an undirectional graph, where each vertex is an edge group. Each vertex is weighted by the weight improvement of this edge group over base matching. There is an edge $e = (EG_i, EG_j)$ in the conflict graph when EG_i and EG_j are conflict with each other. We can detect if two edges groups conflict with each other by checking if they share some common vertices. Then the problem of finding maximum weight co-existing edge groups is equivalent to find the independent set with maximum weight over the conflict graph FG. [2] This problem is called maximum independent set problem, which has been proved to be NP-complete [GJ79].

4.4.3.4 Heuristics algorithms

Previous studies have also shown that the maximum independent set problem is equivalent to the maximum clique problem and the graph coloring problem. These problems are all NP-hard [GJ79]. Many heuristic algorithms have been studied to find approximated solution for maximum clique and independent set problem, such as sequential greedy heuristics, local search heuristics, simulated annealing, neural network and genetic algorithms [FP01]. More specifically, Aarts and Korst [AK89] suggested a simple algorithm to solve the maximum independent set problem using simulated annealing. Homer and Peinado [HP96] compared the performance of several heuristics and find simulated annealing quite effective over various types of large graphs.

The simulated annealing algorithm proposed in previous studies uses a penalty function approach. The solution space is the set of all possible subsets of vertices in the graph G and the problem is formulated as one of maximizing the cost function $f(V') = |V'| - \lambda |E'|$, where $|E'|$ is the number of edges in $G(V')$ and λ is a weighting factor larger than 1. The

[2] In graph theory, an independent set is a set of vertices in a graph, no two of which are adjacent. A maximum independent set is a largest independent set for a given graph G.

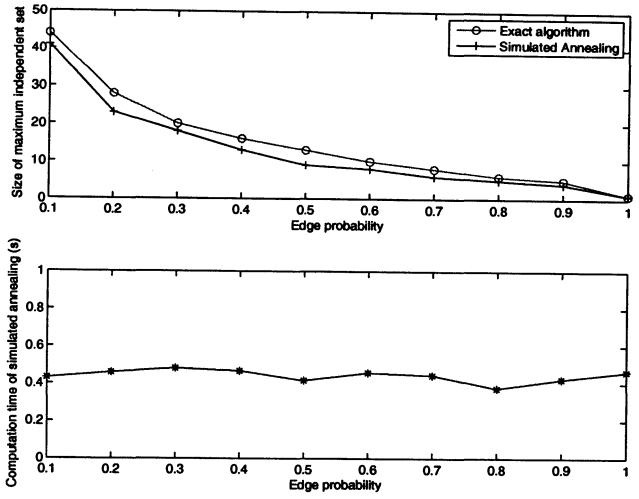


Figure 4.4 : Simulation result: performance of simulated annealing algorithm in finding approximated maximum independent sets

allowed state changes from one solution to another are adding to V' or deleting from V' a random chosen vertex.

We perform a simulation study to evaluate the performance of simulated annealing algorithm over large conflict graphs. We compare the simulated annealing algorithm with the naive exact algorithm on the size of maximum independent set. In this experiment, we generate random conflict graphs with 500 nodes. Each node represents one correlated edge group. We vary the edge probability between any two nodes from 0.1 to 1.0 to test conflict graph with different connectivity densities. The weight of nodes are all set to 1, so that the overall weight of an independent set is the cardinality of this set. Figure 4.4 shows the size of maximum independent set found by the exact algorithm and simulated anneal algorithm, and the computation time of simulated annealing. For all the random graphs with different densities, simulated annealing algorithm can find an independent set whose size is very close to the optimal maximum independent set. The simulated annealing algorithm is quite efficient, which only takes around 400ms on large graphs with 500 nodes (while the exact algorithm takes several days to compute). Our simulation study does confirm that simulated annealing algorithm is effective and efficient in finding a good approximation of maximum

independent set.

4.4.4 Traffic monitoring and analysis

In previous section, we have studied the algorithms to configuration optical circuits supporting correlated traffic. We assume the correlated flows and traffic demands are known during our algorithm design. However, in reality, we may not know the traffic demand and location of correlated flows. The goal of "Analyze" phase is to extract the traffic demand information and application semantics that are needed to perform optical circuit configuration, which include but not limited to acquire more accurate traffic demand, detecting bursty flows and infer correlated flows. In this section, we give a preliminary discussion of the traffic analysis.

4.4.4.1 Detecting bursty flows

We propose a simple heuristic in the *Analyze* phase of our prototype to eliminate the undesirable effect of the bursty flows. The idea is to identify the likely bursty flows and filter them out from the inter-rack traffic matrix before using that as the input for scheduling of circuits. We do this by maintaining idle and active counters for each flow that denote the number of control loop cycles for which the flow has been idle or active, respectively. According to our heuristic, a flow is considered active while it is sending data across the network and we classify a flow as non-bursty if its active counter is greater than a configurable threshold value. If a flow has been idle for a specific number of cycles, we again reset its idle and active counters in order to account for the changing nature of flow, e.g. when a previously stable flow becomes bursty. We evaluated the effectiveness of this approach by repeating the experiment from Section 4.3.1 for demonstrating the effect of bursty flows. Our results confirmed that our prototype avoided the unnecessary circuit flapping due to incorrect demand estimation for the bursty flows and the circuit link was utilized to its full capacity by the long foreground flow.

4.4.4.2 Measuring natural traffic demand

Configuration of optical circuits relies on accurate measurement of application traffic demand in data centers. However, as we have shown in previous sections, c-Through and Helios estimate traffic demand using buffer occupancy and flow counters, which could lead to sub-optimal performance due to inaccurate demand estimation.

If we think about the traffic demand estimation problem further, the major difficulty is, to achieve best circuit configuration, we need to know the application traffic demand in the ideal scenario where there is no bottleneck in the network. We call it "natural traffic demand". Neither buffer occupancy nor flow counters alone represent the natural traffic demand of applications. That is the source of inaccuracy for c-Through and Helios.

We argue that we should combine buffer occupancy on end hosts and flow counters on switches to measure the natural traffic demand of applications. Assuming there is unlimited amount of memory on end hosts used for socket buffers, the natural traffic demand of applications is composed of two parts: the data that has been transmitted by the network, and the data buffered in socket send buffers. For a cross-rack path from rack i to rack j, $natural\_demand(i,j) = buffer\_data(i) + flow\_throughput(i,j)$. We can measure the amount of buffered data using the socket buffer occupancy on all the end-hosts in rack i. The flow throughput on a path can be measured from the flow counter readings on switches.

4.4.4.3 Inferring correlated traffic

We consider this problem in two cases with different assumptions about application knowledge.

Case 1: location known, traffic demand unknown

In the first case, we assume the location of correlated traffic are known, but the traffic demand and its variations are unknown. This is useful for the cases where we can collect the location of correlated applications from data center operator or job manager, but data center operators cannot report traffic demand on correlated paths. So the problem is, assuming we know all the correlated edge groups, but we don't know the weight of correlated edges,

how can we measure the weight of correlated edges in each edge group?

As defined in equation 4.1, the weight of correlated edges are defined by the traffic demand on these paths in two cases when they are and are not assigned with optical circuit simultaneously as a group. For an edge e_i in edge group EG_i, we need to measure both $W_b(e_i)$ and $\Delta(e_i)$. One way to measure $W_b(e_i)$ and $\Delta(e_i)$ would be measure the edge throughput of e_i in cases when edge group EG_i is and isn't part of circuit configuration. In a naive solution, we can measure the demand of edges in edges group EG_i in one round by configuring EG_i with optical circuits and monitoring the throughput of all the edges in EG_i. If there are N edge groups, we need N rounds to measure the demand of all the edge groups.

In a better solution, we can measure the traffic demand of multiple edge groups in one round if these edge groups are not conflict with each other. We can use the following algorithm to find the co-existing edge groups and measuring their demands simultaneously.

1. Since we know the locations of edge groups, we can construct a conflict graph CG for all the edge groups.

2. At the beginning of each round, compute a maximal independent set MIS from the conflict graph CG. [3]

3. Configure the edges groups in MIS with optical circuits and measure the bandwidth demand on these edge groups simultaneously. Configure the remaining paths using default Edmonds' algorithm.

4. At the end of each round, remove edge groups in MIS from the conflict graph CG. Continue the measurement in the next round until all the edge groups have been removed from the conflict graph.

[3] A maximal independent set is an independent set such that adding any other vertex to the set forces the set to contain an edge.

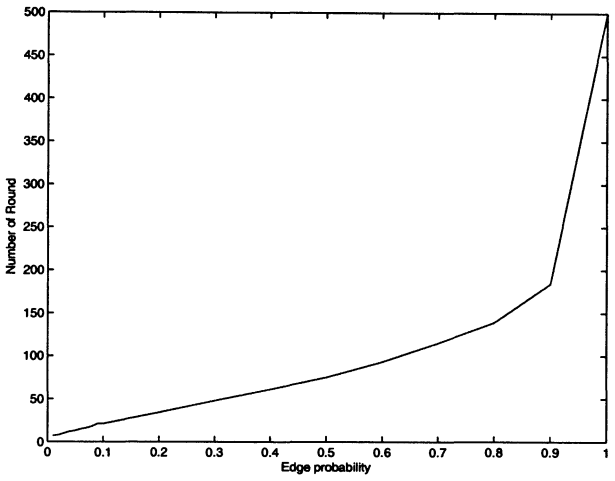


Figure 4.5 : Simulation result: the number of rounds needed to measure 500 edge groups with different conflict probability

To achieve the best possible performance, we would compute the maximal independent set with maximum cardinality in each round. But finding the maximal independent set with maximum cardinality is NP-hard as we discussed earlier. However, we can enumerate all the maximal independent set using Bron-Kerbosch algorithm [BK73]. So we use a fast heuristics to find a independent set with maximum cardinality from the first 10 maximal independent sets reported by Bron-Kerbosch algorithm.

We use a simulation implementation to study how the algorithm performs with a large number of edge groups. We generate random conflict graphs with 500 edge groups and different edge probabilities from 0.01 to 1.0. Higher edge probability means more conflicts among edge groups. Figure 4.5 shows the number of rounds needed to measure all the edge groups with different edge probabilities. From this figure, even for dense conflict graphs with 0.1 edge probability, we only need 21 rounds to measure all the 500 edge groups. The result suggests significantly speedup by measuring non-conflicting edge groups simultaneously.

Case 2: location unknown, traffic demand unknown

In the second case, neither the traffic demand nor the location of correlated flows are

Src	Dst	Appearance	Thr_avg	Thr_var	Surge_config

Table 4.1 : An example of edge table for correlated edge detection

known. This is the most difficult scenario where no application information is reported to the circuit manager. We have to rely on some passive heuristics to infer the correlated traffic in a best effort manner.

The input data we can collect is all the recent circuit configuration and circuit throughput results. Our goal here is to identify the correlated edge groups by observing the traffic throughput measured in recent circuit configuration rounds. Given our modeling of correlated edge groups, we may detect a group of edges as being correlated if we observe significant throughput surge when these edges are configured with optical circuit simultaneously. One particularly idea is, given the recent M rounds of configuration results, we can generate an edge table which collects the statistics about all the edges that have been configured with optical circuits.

Table 4.1 shows the statistics we collect in the edge table. In the table, we use Src and Dst to identify the source and destination rack of an edge. $Appearance$ is how many time this edge has been configured with optical circuit in recent M rounds. Thr_avg and Thr_var is the average and variance of traffic throughput measured on this edge when it appeared in recent configuration. $Surge\_config$ stands for the round number of the configuration where we observe throughput surge on this edge. There are different rules to define "throughput surge". One example rule we have used is that we call an edge has a throughput surge when its throughput is higher than $Thr\_avg + 2 * Thr\_var$.

We implement a circuit configuration simulator to analysis the behavior of edge group detection algorithm. The simulator simulates the circuit configuration in a round-by-round manner. At the beginning of simulation, we generate a set of random edge groups(50 edge groups with 2 edges in each group, the traffic demand on these edges will increase 5Gbps when they are configured simultaneously.). In each round, we generate a synthetic traffic

demand matrix (random traffic demand with average of 1Gbps and variance of 100Mbps), and compute the circuit configuration using Edmonds' algorithms. Then at end of each round, we measure the circuit throughput as the reported traffic demand in demand matrix. The simulator always keep the record of recent M round of configuration and circuit throughput information. We implement the edge group detection algorithm discussed above in the simulator, and find that the detection algorithm can successfully detect the correlated edge groups if they do appear in the recent M configuration. But they edge group detection does not improve the overall circuit throughput performance significantly.

The difficulties are mainly from two parts. First, given no knowledge about the location of potential correlated paths, the chance to even observe the traffic demand surges of correlated paths are very small. For the all-to-all correlated edges we have discussed, the traffic demand on these edges will be increased only when they are all configured with optical circuits. So we can observe the correlation of these paths only when they are selected into circuit configuration simultaneously in one round. By default, the circuit configuration is based on Edmonds' algorithm. The chance that random correlated paths are all selected by Edmonds' algorithm is very small in a large data center. For example, for a random traffic demand matrix with 1000 racks, the chance that two particular edges simultaneously configured with optical circuits is only 0.0001%.

Second, the dynamics in data center traffic make it hard to distinguish random traffic changes and the traffic rate changes caused by application correlation. Even when we do observe the throughput changes over correlated paths, we still need to differentiate them from random traffic changes to reduce the false alarm of correlated traffic.

Based on our analytical findings, we argue that when no information if provided about application level information, the heuristics to infer correlated edge groups can only be used in a best-effort manner. It cannot provide guarantees on overall performance improvement. To achieve good performance for important applications, data center operators will have to register the location of these applications to the circuit controller. Circuit manager can collect the location of correlated edges for these applications, and use the algorithms we

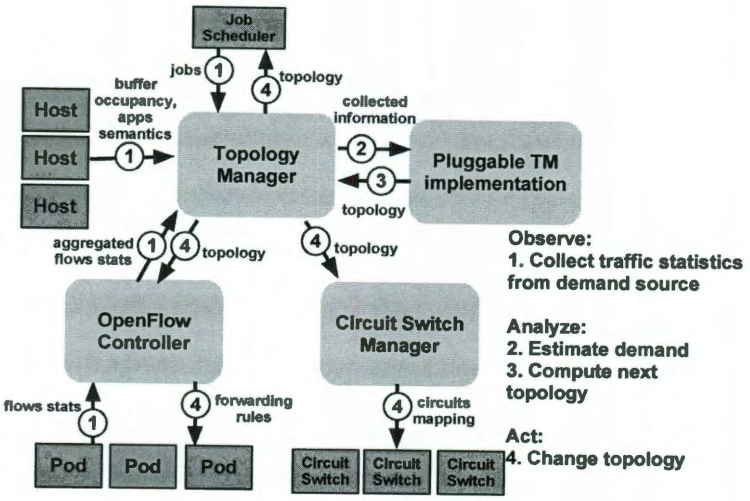


Figure 4.6 : The control loop

have discussed previously to assigned circuit for these applications.

4.4.5 OpenFlow based control system

We are prototyping a OpenFlow based system to support the *Observe-Analyze-Act* framework. Figure 4.6 shows the control loop and the main software components involved. Our prototype includes three main components: Topology Manager, OpenFlow Controller application and Circuit Switch Manager. The Topology Manager is the heart of the system, coordinating with other components to collect information for the Observe phase. To collect input from the network, we leverage the existing OpenFlow API using an OpenFlow Controller system (e.g., as NOX, Ethane, Beacon, or Maestro). We have implemented the Analyzer as a pluggable piece of software in the Topology Manager; this allows us to separate policy from mechanism and evaluate different optimization goals and algorithms. Based on the output of analysis, during the Act phase, the Topology Manager provides the new topology to the Circuit Switch Manager to configure the optical links, to the OpenFlow Controller application, and to the application job scheduler. The OpenFlow Controller configures the switches to forward traffic over appropriate links. The job scheduler

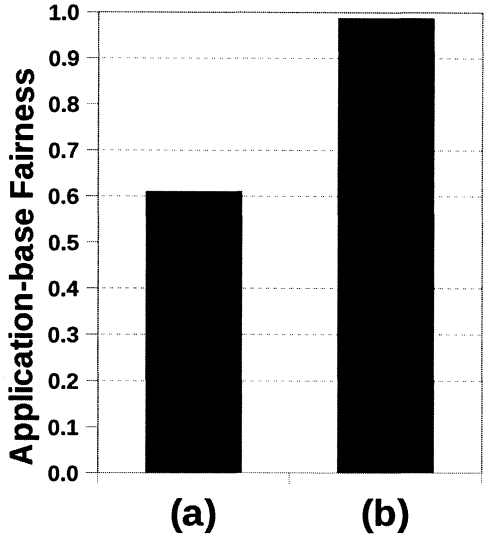


Figure 4.7 : Application fairness based on Jain's fairness metric in (a) Helios/c-Through vs. (b) the proposed framework

can potentially use this information to schedule jobs that can take advantage of the resulting underlying network.

Our experience in using an OpenFlow controller for this prototype has been positive: it provides sufficiently fine-grained control over flow placement on the optical links, and it has allowed us to implement different hashing policies and flow management decisions that achieve better sharing among applications in the cloud. Our experience meshes well with prior work that used OpenFlow to create a unified control plane for IP/Ethernet and optical circuit-switched networks for long-haul backbone networks [DPS+10].

A Usage example: application level fairness

Although many design details remain to be flushed out, we can already realize some flexible control over optical circuits using our fine-grain control framework. In this section, we demonstrate a simple example application level circuit sharing.

The HyPaC system should be able to support a rich mix of a variety of applications. It then becomes imperative that it provides with certain QoS or fairness guarantees for the different applications in the system. Consider the following scenario in a HyPaC sys-

tem: Application A has 18 flows sharing a superlink of 20 Gbps capacity (consisting of two 10 Gbps circuit links) with application B, which sends 2 flows over the shared superlink. Since Helios/c-Through implement naive hashing of flows across the circuits in the superlink, each application gets a share of the optical link bandwidth proportional to the number of flows they have. This means that even though Helios/c-Through provide flow level fairness in the network, their flow management leads to unfair utilization of the superlink at the application level. We overcome this deficiency with our proposed OpenFlow integrated prototype. With more flexible ways to assign flows to the circuit, our prototype can guarantee different fairness objectives including application based fairness by first classifying flows into applications and then assigning flows such that application level fairness is achieved. Figure 4.7 shows how this modification to the scheduling algorithm increases the Jain's fairness index [JCH84] for applications A and B compared to the naive Helios/c-Through policy. Importantly, no external input is required from the applications or network administrators to provide this application-level fairness. The controller can use port numbers as a coarse-grained way to classify flows into applications for use in subsequent circuit selection.

We emphasize that the point of these preliminary designs is not to show that they are the final answer to the challenges we described—indeed, we are fairly certain they are *not*. Instead, we believe they demonstrate that the framework we propose helps pave the way to future solutions in this area.

4.5 Discussion

The flexible control framework allows us to explore design choices in managing hybrid network in datacenters. We hope to investigate these components in more detail in the future, and hope that this paper spurs others to do likewise.

Traffic analysis with application semantics: Efficient traffic engineering in HyPaC networks relies on precise and detailed traffic analysis. In addition to burst flow detection, the traffic analyzer should detect any ill-behaved flows that may disrupt the network

configuration or consume more bandwidth than their fair-share. This feature is important for system reliability. To support controlled sharing among mixed applications, a second challenge is learning the salient application characteristics, either by inferring them using traffic analysis or by devising a cluster-wide abstraction for applications to provide such information.

Traffic analysis requires thorough, but efficient, monitoring and aggregation of network traffic and status. We can classify application flows in controlled datacenter environments based on flexible header signatures, especially in OpenFlow deployments. Bursty flows can be detected by analyzing the flow counters on each switch. We can reveal correlated flows by analyzing the dependency of flow rate changes when the network is reconfigured. Given the frequent reconfiguration of circuits, the system level challenge is to perform traffic analysis accurately and efficiently with a limited number of flow samples, a problem similar to many in streaming databases.

Circuit scheduling: The objective of circuit scheduling is a combination of performance and fairness. For same-priority applications, the objective is to maximize the overall throughput of the optical circuits. For applications with different priorities, appropriate sharing policies must be defined (for example, strict priority, max-min fairness). Based on application semantics from the traffic analysis, different sharing policies can be implemented in the controller. The remaining questions are how the scheduling framework coordinates and balances the trade-offs between performance and sharing objectives.

Flow aggregation: An OpenFlow-based control framework enables per-flow based forwarding decisions on both electrical and optical links. However, due to limited number of flow table entries on ToR switches, it is not feasible to install a forwarding rule for each flow and flow aggregation becomes necessary. Medium size data centers could have hundreds of thousands of flows generated every second, but existing OpenFlow switches can only support a few thousand table entries.

This leads to a new optimization objective: minimizing the number of forwarding entries required to configure the HyPaC network while preserving the desired traffic control

policies. A possible solution is to group all the correlated flows and flows with the same priority into a single flow group, and compute the most concise *wildcard* rule associated with that flow group. It might be possible to leverage similar algorithms proposed in related work [WBR11].

4.6 Summary

In this chapter, we further explore the control of optical circuits in a cloud data center shared by many heterogeneous applications. We identify a set of challenges in adopting optical circuit switches in real data center environments and propose a "observe-analyze-act" control framework to achieve flexible, fine-grain and responsive control of optical circuits among multiple applications. We discuss the design of scheduling algorithms to support traffic dependencies and build an OpenFlow-based circuit control system. Preliminary evaluation has demonstrated the advantage of new control framework to achieve flexible sharing policies among applications.

Chapter 5

Networking Performance in Virtualized Data Centers

5.1 Motivation and Overview

Cloud services allow enterprise class and individual users to acquire computing resources from large scale data centers of service providers. Users can rent machine instances with different capabilities as needed and pay at a certain per machine hour billing rate. Despite concerns about security and privacy, cloud service attracts much attention from both users and service providers. Recently, many companies, such as Amazon, Google and Microsoft, have launched their cloud service businesses.

Most cloud service providers use machine virtualization techniques to provide flexible and cost-effective resource sharing among users. For example, both Amazon EC2 [Ama] and GoGrid [GoG] use Xen virtualization [BDF+03] to support multiple virtual machine instances on a single physical server. Virtual machine instances normally share physical processors and I/O interfaces with other instances. It is expected that virtualization can impact the computation and communication performance of cloud services. However, very few studies have been performed to understand the characteristics of these large scale virtualized environments.

In this chapter, we present an empirical measurement study on the end-to-end networking performance of the commercial Amazon EC2 cloud service, which represents a typical large scale data center with machine virtualization. The focus of our study is to characterize the networking performance of virtual machine instances and understand the impact of virtualization on the network performance experienced by users.

Observations: We measure the processor sharing, packet delay, TCP/UDP throughput and packet loss properties among Amazon EC2 virtual machine instances. Our study

systematically quantifies the impacts of virtualization and finds that the magnitude of the observed impacts are significant:

1. We find that Amazon EC2 small instance virtual machines typically receive only a 40% to 50% share of the processor.

2. Processor sharing can cause very unstable TCP/UDP throughput among Amazon EC2 small instances (discussed in section 5.2). Even at the tens of millisecond time granularity, the TCP/UDP throughput experienced by applications can fluctuate rapidly between 1 Gbps and zero.

3. Even though the data center network is not heavily congested, we observe abnormally large packet delay variations among Amazon EC2 instances. The delay variations can be a hundred times larger than the propagation delay between two end hosts. We conjecture that the large delay variations are caused by long queuing delays at the driver domains of the virtualized machines.

4. We find that the abnormally unstable network performance can dramatically skew the results of certain network performance measurement techniques.

Implications: Our study serves as a first step towards understanding the end-to-end network performance characteristics of virtualized data centers. The quantitative measurement results from this study provide insights that are valuable to users running a variety of applications in the cloud. Many cloud applications (e.g. video processing, scientific computing, distributed data analysis) are data intensive. The networking performance among virtual machines is thus critical to these applications' performance. The unstable throughput and packet delays can obviously degrade the performance of many data intensive applications. More importantly, they make it hard to infer the network congestion and bandwidth properties from end-to-end probes. Packet loss estimation is an example that will be discussed in section 5.6. The abnormal variations in network performance measurements could also be detrimental to adaptive applications and protocols (e.g. TCP vegas [BP95],

PCP [ACKZ06]) that conduct network performance measurements for self-tuning. Researchers have also recently started to deploy large scale emulated network experiments on cloud services [Clo, RHHR09]. For this use of cloud services, our results point to challenges in performing accurate network link emulation in virtualized data centers. The unstable network performance of cloud services may bias the conclusions drawn from these experiments. Given the observations from our measurement study, many applications may need to be adjusted to achieve optimal performance in virtualized data center environments.

This chapter is organized as follows. In Section 5.2, we introduce the background on Amazon EC2 and the Xen virtualization technique. In section 5.3, we explain our measurement methodology. In Section 5.4, 5.5, 5.6, we describe our measurement results and discuss the reasons behind our observations. In Section 5.7, we discuss the implications of our findings on different classes of cloud applications.

5.2 Background

5.2.1 Xen Virtualization

Xen [BDF+03] is an open source x86 virtual machine monitor which can create multiple virtual machines on a physical machine. Each virtual machine runs an instance of an operating system. A scheduler is running in the Xen hypervisor to schedule virtual machines on the processors. The original Xen implementation schedules virtual machines according to the Borrowed Virtual Time (BVT) algorithm [DC99].

Xen uses *para-virtualization* for network virtualization, which only allows a special privileged virtual machine called *driver domain, or domain 0* to directly control the network devices. All the other virtual machines (called *guest domains* in Xen) have to communicate through the driver domain to access the physical network devices. The way Xen realizes this is that the driver domain has a set of drivers to control the physical Network Interface Cards (NIC), and a set of *back-end interfaces* to communicate with *guest domains*. The back-end interfaces and physical drivers are connected by a software bridge inside the

kernel of the driver domain. Each *guest domain* has a customized virtual interface driver to communicate with a back-end interface in the driver domain. All the packets sent from *guest domains* will be sent to the driver domain through the virtual interfaces and then sent into the network. All the packets destined to a *guest domain* will be received by the driver domain first, and then transferred to the *guest domain.*

5.2.2 Amazon Elastic Cloud Computing (EC2)

Amazon EC2 is a component of Amazon's Web Services (AWS), which allows users to rent computers to run computer applications in the Amazon EC2 data center. Amazon EC2 uses the Xen virtualization technique to manage physical servers. There might be several Xen virtual machines running on one physical server. Each Xen virtual machine is called an *instance* in Amazon EC2. There are several types of instances. Each type of instance provides a predictable amount of computing capacity. The small instance is the primary instance type, which is configured with 1.7GB memory, 1 EC2 compute unit and a 160GB instance storage. According to Amazon, "one EC2 compute unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor." For applications requiring higher computing capacity, Amazon EC2 provides several high-capacity instances which are configured with 4 to 20 EC2 compute units. The input-output (I/O) capacities of these types of instances are not specified clearly.

Allocated EC2 instances can be placed at different physical locations. Amazon organizes the infrastructure into different regions and availability zones. There are two regions, us-east-1 and eu-west-1, which are located in the US and in Europe respectively. Each region is completely independent and contains several availability zones that are used to improve the fault tolerance within the region. We suspect that each availability zone is an isolated data center which is powered by its own powerline. Different availability zones in the same region are placed very close to each other. The region us-east-1 has three availability zones, us-east-1a, us-east-1b and us-east-1c. The region eu-west-1 has two availability zones, eu-west-1a and eu-west-1b.

5.3 Experiment Methodology

In this section, we introduce the methodology of our measurement study. We first explain the properties we measure in our experiments, and the methodology we use to measure them.

5.3.1 Properties and Measurement Tools

Processor Sharing: Since each Amazon EC2 instance is a Xen virtual machine, an immediate question users may ask is "how does Amazon EC2 assign physical processors to my instance? Is there any processor sharing?" To answer this question, we use a simple CPUTest program to test the processor sharing property of EC2 instances. This program consists of a loop that runs for 1 million times. In each iteration, the program simply gets the current time by calling gettimeofday() and saves the timestamp into a pre-allocated array in memory. When the loop finishes, the program dumps all the saved timestamps to the disk. Normally, if the program is executed continuously, all loop iterations should take a similar amount of time. However, virtual machine scheduling can cause some iterations to take much longer than the others. If the instance is scheduled off from the physical processor, we should observe a gap in the timestamp trace. Since context switching among user processes can also cause a gap in the timestamp trace, we always run the CPUTest program as the only user process in our processor sharing experiments to minimize the impact of context switching. Therefore, from the timestamp trace of this program, we can estimate the processor sharing property of EC2 instances.

Packet round-trip delay: Given an instance pair, we use ping to measure the packet round-trip delay (or round-trip time, RTT) between them. To also measure delay variations, we send 10 ping probes per second, and continuously collect 5000 round-trip delay measurements.

TCP/UDP throughput: We developed two programs TCPTest and UDPTest to measure the TCP and UDP throughput that can be achieved by applications running on Amazon EC2 instances. The UDPTest tool has a sender and receiver. The sender reads data from

a buffer in memory and sends it as UDP packets. Since Amazon EC2 instances are Xen virtual machines, the UDP packets are sent to network through Xen driver domain. The communication between Xen driver domain and guest domain is done by copying data from memory to memory. If UDP sender sends as fast as possible, it will burst data at a very high rate to the driver domain. A lot of traffic will be dropped when Xen driver domain cannot send them out in time. Therefore, in our UDPTest tool, the sender controls the sending rate to 1Gbps by adding small idle intervals between every 128KB of data. We set the sending rate to 1Gbps because according to our experiences, the Amazon EC2 instances are configured with Gigabit network cards. The UDP/IP packet size is 1500 bytes (i.e. the MTU of Ethernet) and the socket buffer size is 128KB. The receiver simply receives the UDP data and calculates the UDP throughput. The TCPTest tool also has a sender and a receiver. The sender reads data from a buffer in memory and sends data via a TCP connection to the receiver. The receiver also simply receives the data and calculates the TCP throughput. The TCP maximum send and receive window sizes are set to 256KB. From our experience, most of the RTTs among Amazon EC2 instances are below 0.5 ms. Therefore, if the network allows, end host could achieve throughput higher than 4Gbps with this window size. With this setting, we make sure that the TCP flow can saturate the network interface.

Packet loss: We use the Badabing tool [SBDR05] to estimate the packet loss among Amazon EC2 instances. Badabing is the state-of-the-art loss rate estimation tool. It has been shown to be more accurate than previous packet loss measurement tools [SBDR05]. Packet loss estimation is considered challenging because packet loss typically occurs rarely and lasts for very short time. Badabing use active probes and statistical estimations to measure the packet loss properties. However, since we are using these tools in a virtualized environment, those estimations may not give us accurate results. We will provide detailed discussion on the packet loss estimation results in section 5.6.

5.3.2 Instance Type Selection

Amazon EC2 provides different types of instances for users. Our measurement experiments are mainly based on Amazon EC2 small instances and high CPU medium instances (also called medium instances). Small instances are the default instances in Amazon EC2 and they compete for physical processor resources, which creates an interesting environment for studying the impact of virtualization on network performance. High CPU medium instance is one type of high-capacity instances in Amazon EC2. Based on Amazon EC2 documents, the high-capacity instances are configured with multiple virtual cores (2 for high CPU medium instances). Each virtual core represents a CPU core that is visible inside a virtual machine. It is expected to have no processor competing among high-capacity instances. We choose medium instances as comparison with small instances to study the cases with and without processor sharing among virtual machines.

5.3.3 Large Scale Experiment Setup

We deploy large scale experiments to evaluate the system wide networking performance of Amazon EC2 instances. We set up a spatial experiment to evaluate how the network performance varies for instances at different network locations. We set up a temporal experiment to evaluate how the network performance varies on a given instance over a long time period. All the large scale experiments are deployed in the us-east-1 region. To eliminate the potential impacts from different kernel versions, we use the same OS image ami-5647a33f on all the instances.

Spatial experiment: In the spatial experiment, we request 250 pairs of small instance and 50 pairs of medium instances from each of the three availability zones us-east-1a, us-east-1b and us-east-1c. Within each availability zone, the instances are requested and measured in a round by round manner. In each round, we request a pair of instances, measure them and release them. Since we don't know the instance allocation strategy of Amazon EC2, we check the network mask of all the instances to validate that the requested instances are at different network locations. According to the network mask, the instances

we have chosen cover 177 different subnets in Amazon EC2 data centers. For each instance pair, we measure the processor sharing using CPUTest on both instances. We also measure the network properties between the two instances, including delay, TCP/UDP throughput and packet loss. To avoid interference between different measurement programs, we run the programs one by one sequentially. Since the TCP/UDP throughput measurement programs are more intrusive to the network, we limit the amount of data transmitted in each test to 800 MB, which corresponds to roughly 10 seconds of measurement time. We run the Badabing tool for one minute to estimate the packet loss property for an instance pair. Since all the instance pairs in the same availability zone are measured sequentially, the measurement traffic of different instance pairs will not interfere with each other.

Temporal experiment: In the temporal experiment, we choose two small instance pairs and one medium instance pair in each of the three availability zones (us-east-1a, us-east-1b and us-east-1c). For all the nine instance pairs, we measure their processor sharing and network performance continuously for 150 hours. The measurements are done in a round by round fashion. Within each availability zone in each round, we measure the processor sharing, RTT, TCP/UDP throughput and packet loss of the three instance pairs one by one. The settings of all the measurement tools are the same as in the spatial experiment. The time interval between two adjacent rounds is set to 10 minutes. We arrange all the experiments inside the same availability zone sequentially to avoid interference between measurement traffic.

5.4 Processor Sharing

We use our CPUTest program to test the processor sharing on small and medium instances in our spatial and temporal experiments. We first present a typical CPUTest timestamp trace observed on small and medium instance in Figure 5.1. As illustrated in Figure 5.1, when the CPUTest program is run on a non-virtualized machine or a medium instance, the timestamp traces produced indicate the CPUTest program achieves a steady execution rate with no significant interruption. However, the timestamp trace of the small instance shows

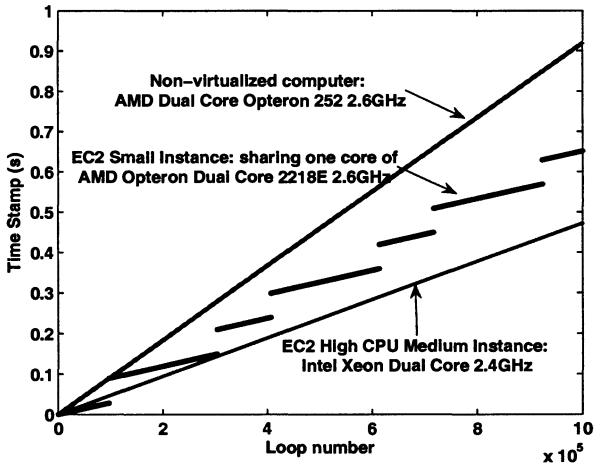


Figure 5.1 : CPUTest trace plot

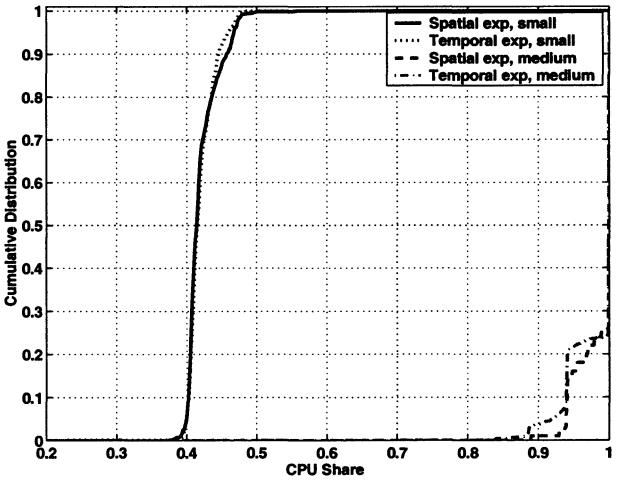


Figure 5.2 : The distribution of CPU share

very obvious scheduling effects. When the CPUTest program is run on a EC2 small instance, periodically there is a big timestamp gap between two adjacent loop iterations. The timestamp gaps are on the order of tens of milliseconds. In each iteration of the CPUTest program, the program only retrieves the current time and saves it to memory; there are no I/O operations. Since CPUTest is the only user program running on the instance, there

shouldn't be frequent context switching between user programs. Therefore, the logical reason that explains the observed execution gap is virtual machine scheduling. The large timestamp gaps represent the periods when the instance running the CPUTest program is scheduled off from the physical processor.

In the CPUTest timestamp trace on an EC2 small instance, when the CPUTest program is running on the processor, one loop iteration normally takes less than 3 us. If one loop iteration takes more than 1 ms, we treat this time gap as a schedule off period. We define CPU sharing as the percentage of CPU time an instance gets from the Xen hypervisor. By searching through the timestamp traces produced by the CPUTest program, we can estimate the CPU sharing of EC2 instances. Figure 5.2 shows the distribution of CPU sharing estimation of small and medium instances in both spatial and temporal experiments. From this graph, we can see that small instances are always sharing processors with other instances. For almost all the cases, small instances always get 40% to 50% of the physical CPU sharing. We suspect Amazon EC2 uses strict virtual machine scheduling policy to control the computation capacity of instances. Even there is no other virtual machines running on the same server, small instances still cannot use more than 50% of the processor. On the other hand, medium instances get 100% CPU sharing for most of the cases. There are only 20% of the cases where medium instances get 95% of the CPU sharing, which might be caused by the context switch between the CPUTest program and kernel service processes.

Note that the scheduling effect observed by our CPUTest program is only typical for CPU intensive applications since it does not have any I/O operations during the test period. I/O intensive applications may have different scheduling pattern. However, our results do confirm that processor sharing is a wide spread phenomenon among EC2 small instances, whereas medium instances do not competing processors with other instances.

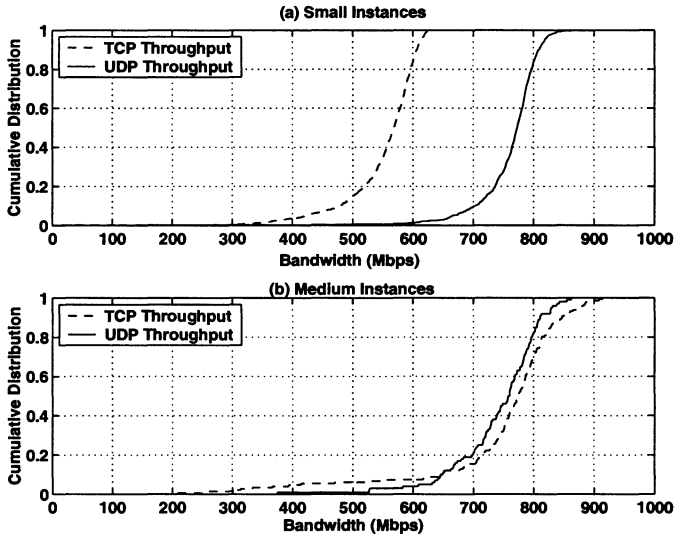


Figure 5.3 : The Distribution of bandwidth measurement results in spatial experiment

5.5 Bandwidth and Delay Measurement

In this section, we discuss the bandwidth and delay measurement results of Amazon EC2 instances observed in our experiments.

5.5.1 Bandwidth Measurement

Measurement Results: In the spatial experiment, we measured the TCP/UDP throughput of 750 pairs of small instances and 150 pairs of medium instances at different network locations. In the temporal experiment, we measured the TCP/UDP throughput of 6 pairs of small instances and 3 pairs of medium instances continuously over 150 hours. Figure 5.3 shows the cumulative distribution of TCP/UDP throughput among small and medium instances in the spatial experiment. From these results, we can see that Amazon EC2 data center network is not heavily loaded since EC2 instances can achieve more than 500 Mbps TCP throughput for most the cases. More importantly, we can make an interesting observation from this graph. Medium instances can achieve similar TCP and UDP throughput. The median TCP/UDP throughput of medium instances are both close to 760 Mbps. However,

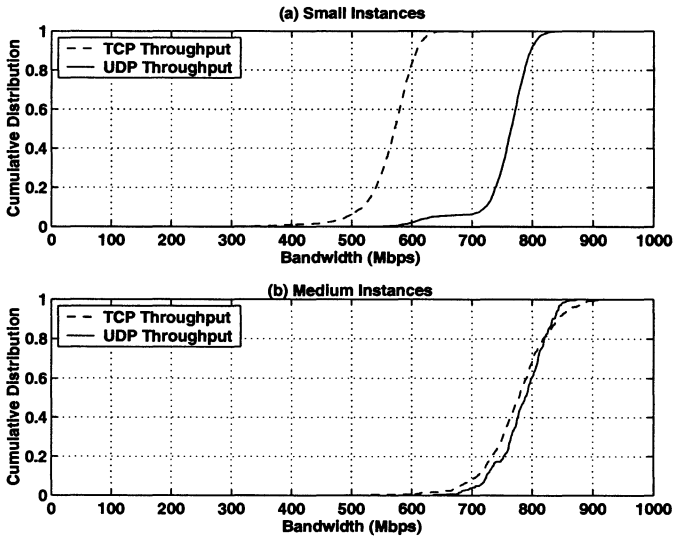


Figure 5.4 : The distribution of bandwidth measurement results in temporal experiment

the TCP throughput of small instances are much lower than their UDP throughput. The median UDP throughput of small instances is 770 Mbps, but the median TCP throughput is only 570 Mbps. Figure 5.4 shows the cumulative distribution of TCP and UDP throughput of small and medium instances over the 150-hour period in our temporal experiment. We observe the same behavior from the results of the temporal experiment. Why is the TCP throughput of small instances much lower than the UDP throughput? We performed more detailed experiments to answer this question.

Discussion: Several factors can impact the TCP throughput results, including TCP parameter settings, packet loss caused by network congestion, rate shaping and machine virtualization. In our experiments, the TCP window size we use is 256KB which can achieve 4Gbps throughput if the network allows. Therefore, the low TCP throughput of small instances is not caused by TCP parameter settings. To investigate further, we study the TCP/UDP transmission at a much smaller time scale. In our TCPTest and UDPTest tool, every time when the receiver receives 256KB of data, it computes a throughput for the recent 256KB data transmission. Figure 5.5 demonstrates the fine-grain TCP and UDP

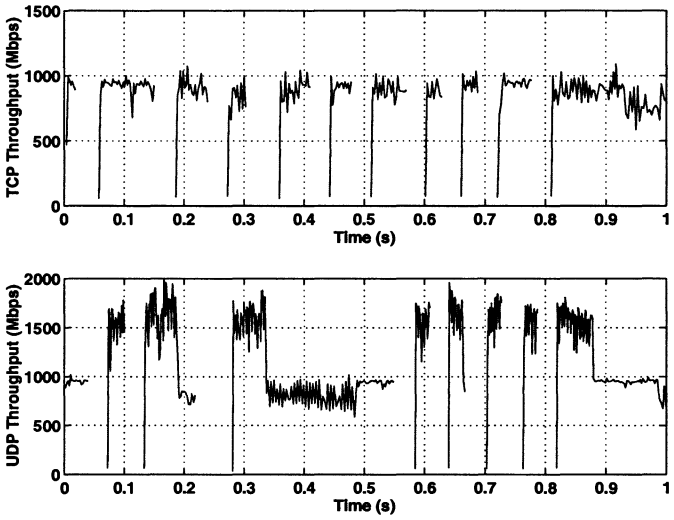


Figure 5.5 : TCP and UDP throughput of small instances at fine granularity

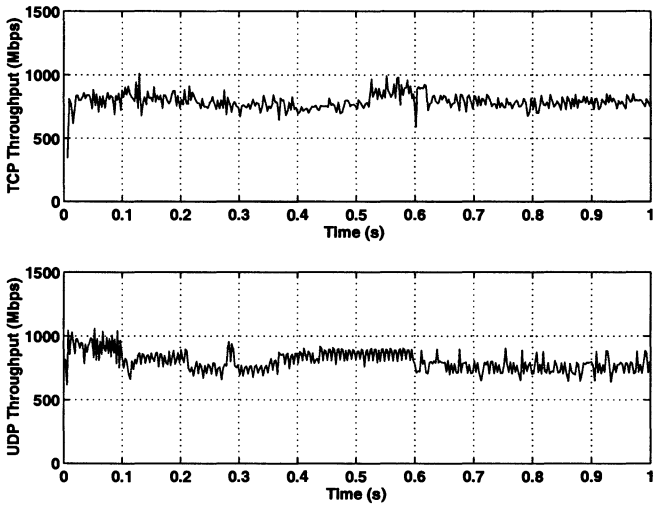


Figure 5.6 : TCP and UDP throughput of medium instances at fine granularity

throughput of a typical small instance pair for a 1-second transmission. We consistently observe the same transmission pattern on all the small instances. To make the results clearly visible, we only pick one small instance pair and plot the throughput for a 1-second period.

First, we discuss the behavior of TCP transmission. We observe the drastically unstable

TCP throughput switching between full link rate at near 1 Gb/s and close to 0 Gb/s. From these transmission patterns, the relatively low average TCP throughput does not appear to be caused by any explicit rate shaping in Xen because typical rate shapers (e.g. a token bucket rate shaper) would not create such oscillations.

By looking at the TCPDUMP trace of the TCP transmission, we find that during the very low throughput period, no packet is sent out from the TCP sender. The quiet periods last for tens of milliseconds. The minimum TCP retransmission timeout is normally set to 200 ms in today's Linux kernel [VPS+09]. These quiet periods are not long enough to cause TCP retransmissions. We also confirm that there are no TCP retransmission observed in the TCPDUMP trace. This result tells us that the periodic low TCP throughput is not caused by packet loss and network congestion because if that is the case, we should observe a large number of TCP retransmissions. Considering the processor sharing behavior observed in our CPUTest experiments, we believe that the quiet periods are caused by the processor sharing among small instances. During these quiet periods, either the TCP sender instance or the receiver instance are scheduled off from the physical processor, therefore no packet can be sent out from the sender.

From Figure 5.5, we can observe a similar unstable UDP throughput on small instances. The difference between UDP and TCP transfers is that, in many cases, after a low throughput period, there is a period where the receiver receives UDP traffic at a high burst rate (even higher than the network's full link rate). That is why UDP throughput is higher than TCP throughput on average. We believe the reason is, during the low UDP throughput periods, the receiver is scheduled off from the processor, but the sender instance is scheduled on. All the UDP traffic sent to the receiver will be buffered in the Xen driver domain. When the receiver is scheduled in later, all the buffered data will be copied from driver domain memory to the receiver's memory. Since the data is copied from memory to memory, the receiver can get them at a much higher rate than the full link rate.

We define a *buffer burst period* by a UDP transmission period during which the receiver continuously receive data at rates higher than the full link rate. Since we set the UDP

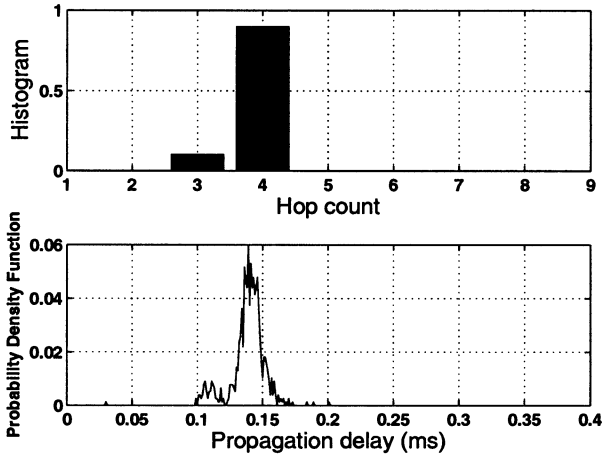


Figure 5.7 : The Distribution of propagation delays and hop count results in spatial experiment

sending rate to 1Gbps, during a *buffer burst period*, the additional amount of data beyond full link rate transfer must come from the Xen driver domain buffer. We call this additional data *buffer burst data*. We can estimate the lower bound of Xen driver domain buffer size by the volume of *buffer burst data*. We analyze the UDP transmission trace of small instance pairs in our 150 hour temporal experiment. We find, in the maximum case, the *buffer burst data* is as high as 7.7 MB. It means that the Xen driver domain buffer can be more than 7.7 MB. The large buffer at Xen driver domain can help reduce the packet loss and improve the average UDP throughput when instances are scheduled off from physical processors.

Figure 5.6 demonstrates the fine-grain TCP/UDP throughput trace for a medium instance pair. Since there is no processor sharing among medium instance pairs, the TCP/UDP throughput is relatively stable. Medium instances can achieve similar UDP and TCP throughput which are decided by the traffic load of the data center network.

5.5.2 End-to-end Delays

Measurement Results: In this section, we discuss the packet delay measurement in our experiments. In our spatial experiment, we measure the packet round trip delay (RTT) of

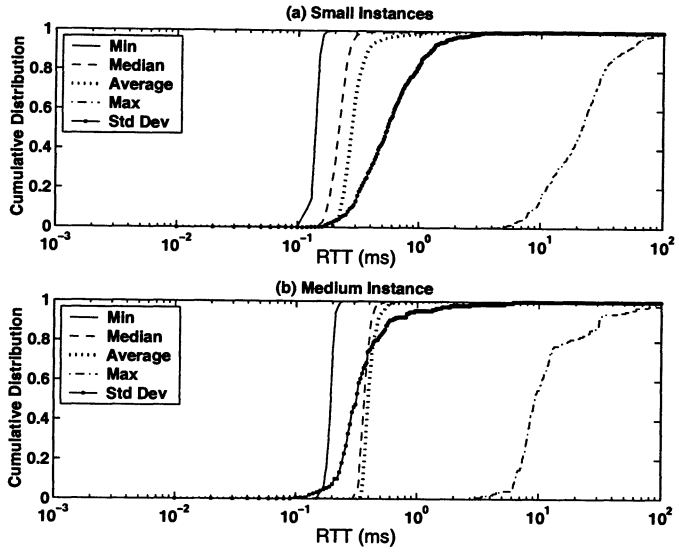


Figure 5.8 : The distribution of delay statistical metrics in spatial experiment

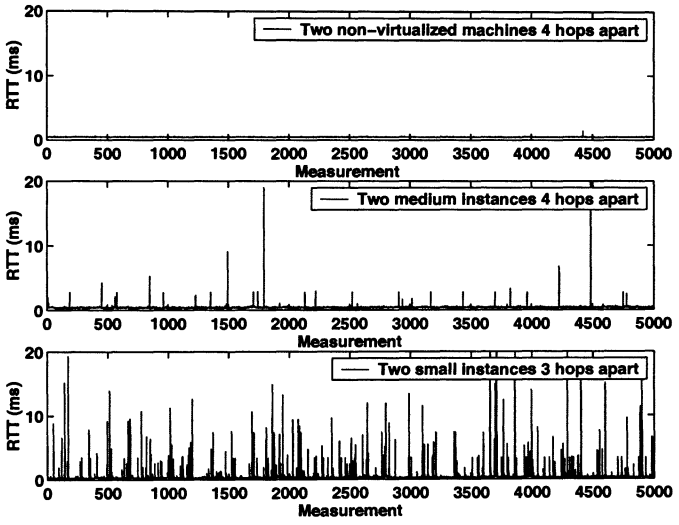


Figure 5.9 : Raw RTT measurements on Amazon EC2 instances and non-virtualized machines in university network

750 small instance pairs and 150 medium instance pairs using 5000 ping probes. Before describing the characteristics of end-to-end delays, we first discuss an interesting observation in our ping measurement results. We consistently observe very large delays (hundreds

of ms) for the first several ping probe packets over all the instance pairs in our spatial experiment. We compare the ping RTT results with the RTTs measured by UDP probe packets. We find that the UDP probe RTTs have the same characteristics with the ping RTTs except that the first several UDP probe packets do not have abnormally large delays. By looking at the TCPDUMP trace of the ping packets, we believe the reason for the abnormally large initial ping RTTs is that every time ping packets are initiated between an instance pair, the first several ping packets are redirected to a device, perhaps for a security check. The routers can forward ping packets only after the security check device allows them to do so. The large delays of the first several ping packets are caused by the buffer delay at the security device. Therefore, in our RTT characteristics analysis, we remove the RTT measurement results of the first 50 ping packets to eliminate the impact of this security check on our delay measurements.

We analyze several characteristics of RTTs among EC2 instances. First, we estimate the propagation delays between instance pairs using the minimum RTTs observed in ping probes. In Figure 5.7, the bottom graph shows the probability distribution of propagation delays for all instance pairs. The propagation delays have a two-peak distribution. The top graph in Figure 5.7 shows the histogram of the hop counts for all the instance pairs. The hop counts are measured using traceroute. From this graph, we can see that in the EC2 data center, instances are very close to each other. All the instance pairs we measured are within 4 hops from each other, and most propagation delays are smaller than 0.2 ms. For all the 900 instance pairs we have measured, the instances are either 3 hops or 4 hops away from each other. This is the reason why we observe a two-peak propagation delay distribution.

For each instance pair, we compute the minimum, median, average, maximum RTTs and the RTT standard deviation from the 4950 probes. Figure 5.8 shows the cumulative distribution of these RTT statistical metrics for small and medium instances (note that the x-axis is in log scale). From this graph, we can see that the delays among these instances are not stable. The propagation delays are smaller than 0.2 ms for most of the small instance pairs. However, on 55% of the small instance pairs, the maximum RTTs are higher

than 20 ms. The standard deviation of RTTs is an order of magnitude larger than the propagation delay and the maximum RTTs are 100 times larger than the propagation delays. The delays of medium instances are much more stable than the small instances. But we still observe that, for 20% medium instance pairs, the maximum RTTs are larger than 10ms. Considering the Amazon EC2 data center is a large cluster of computers that are not spread over a wide area, these large delay variations are abnormal.

As a comparison, we test the RTT and between non-virtualized machines located in our university network and in Emulab. We observe much smaller delay variations on the machines in our university network and in Emulab. For example, for two machines in our university network which are 4 hops away, the minimum/average/maximum RTTs are 0.386/0.460/1.68 ms respectively, and the RTT standard deviation is 0.037 ms. For two machines in Emulab which are connected through a switch, the minimum/average/maximum RTTs are 0.138/0.145/0.378 ms, and the RTT standard deviation is 0.014 ms. For all these non-virtualized machines, the RTT standard deviation is roughly 10 times smaller than the propagation delays. To visualize this difference, we plot the 5000 RTT measurement results for non-virtualized machines, small instances, and medium instances in Figure 5.9. We can clearly see that, RTTs among Amazon EC2 instances have much higher variations than non-virtualized machines. The delay variations among small instances are much higher than that of medium instances.

Discussion: End-to-end delay variation are typically assumed to be caused by the queuing delays on routers when a network is congested. However, in the Amazon EC2 data center, the large delay variations are unlikely to be caused by network congestion. The reasons can be argued as follows. First, we observe very rare packet loss in the ping probes. Among all the instance pairs we have done ping probes, 98% of them did not experience any ping packet loss. The other 2% only experience roughly 1 out of 1000 packet loss. Second, in our bandwidth measurement experiments, all the instances we have measured can achieve at least 500Mb/s TCP throughput. All these results imply that the Amazon EC2 data center network is not congested.

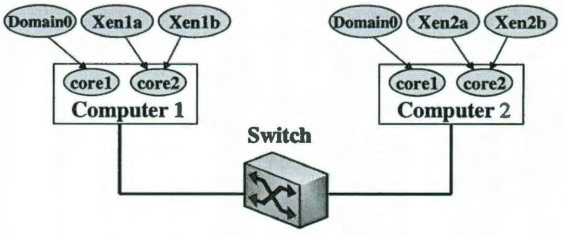


Figure 5.10 : The Configuration of Xen Testbed

Considering the processor sharing and large Xen driver domain buffer observed in previous sections, our conjecture is that the large delay variations among EC2 instances are caused by the long queuing delay at the Xen driver domain. Since small instances are sharing processors with other instances, when the receiver instance is scheduled off, the probe packets will be buffered at the Xen driver domain until the receiver is scheduled on. This long buffering delay causes very high delay variations among small instances. Although medium instances do not share processors, they are still sharing Xen driver domain with other instances. Let us suppose a medium instance A is sharing Xen driver domain with another instance B. Since there is a large buffer in the driver domain, instance B could burst a big trunk of data into the driver domain buffer. In this case, the packet from A could be put into a long queue in Xen driver domain, which leads to relatively long queuing delay. Since packets don't need to wait for the processor scheduling for medium instances, the delay variations on medium instances are generally smaller than small instances.

To cross validate our hypothesis, we set up a Xen testbed in our lab. In the testbed, two computers are connected to each other through a 10Gb/s switch. Both machines are configured with a Dual Core AMD Opteron(tm) 285 processor, 1GB memory and one Gigabit Ethernet card. We install Xen virtual machines on both computers. Each computer runs one Xen domain0 and two Xen virtual machines, where the Xen domain0 is bound to one physical processor core and two guest virtual machines share the other processor core. The configuration of testbed is shown in Figure 5.10. We measure the delays between virtual machine Xen1a and Xen2a using 5000 ping probes with different applications running on

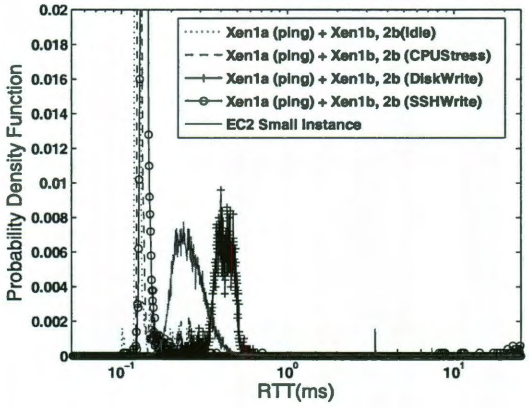


Figure 5.11 : The Delay Distribution of Xen Testbed at Different Application Scenarios

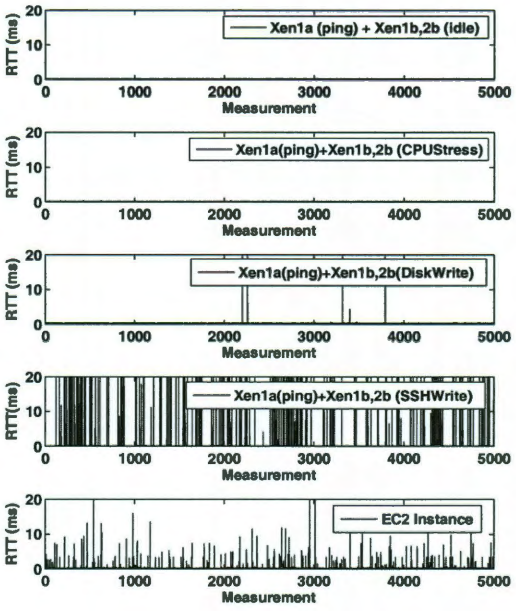


Figure 5.12 : The Demonstration of Raw Delay Measurement Results on Xen Testbed at Different Application Scenarios

virtual machine Xen1b and Xen2b. We test four different application scenarios.

In the first scenario, Xen1b and Xen2b are all idle. Then we introduce some compe-

tition to Xen1a and Xen2a by running CPUStress, DiskWrite and SSHWrite applications on Xen1b and Xen2b. The CPUStress program is just an infinite loop with floating point calculations to consume as many CPU cycles as possible. There are no IO operations. The DiskWrite program continuously writes data from memory into the local disk at 10MB/s. The SSHWrite program continuously writes data from memory to another remote machine outside our testbed through an SSH connection at 10MB/s rate. Figure 5.11 shows the probability distribution of the delays measured on our Xen testbed. To clearly demonstrate the results, we also plot the raw RTT measurements for all the scenarios in Figure 5.12.

From these figures, we can see that, when Xen1b and Xen2b are idle, the delays between Xen1a and Xen2a are very stable. When Xen1b and Xen2b run the CPUStress application, the delays between Xen1a and Xen2a are still stable. This means just processor sharing will not cause large jitters among Xen virtual machines. However, when we introduce some IO competitions into the testbed, the delays between Xen1a and Xen2a become unstable. When Xen1b and Xen2b run the DiskWrite application, the delays between Xen1a and Xen2a have a wide distribution, and in a few cases, the delay jitters can be very large. Furthermore, when Xen1b and Xen2b run the SSHWrite application which encrypts data and sends data into the network. The delays between Xen1a and Xen2a have very large jitters. Although the DiskWrite and SSHWrite applications are not the real applications running on Amazon EC2 instances, the results do show that some level of processor sharing combining with IO sharing among Xen virtual machines can cause very large delay jitters. From these results, we believe the large delay jitters observed between Amazon EC2 instances are very likely to be caused by the processor sharing and IO sharing among Xen virtual machines. Our results are also aligned well with related performance studies performed on Xen testbed. For example, in [OCR08], Ongaro *et al.* performed an I/O performance study on a controlled Xen testbed and also observed similar latency instability caused by the Xen scheduler.

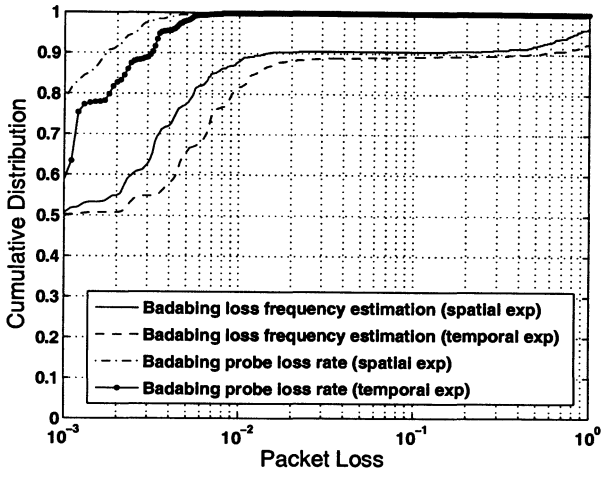


Figure 5.13 : Badabing packet loss results in spatial and temporal experiments

5.6 Packet Loss Estimation

Estimation Results: In this section, we describe the packet loss estimation results observed in our experiments. Badabing estimates the packet loss characteristics of an end-to-end path by estimating if each 5ms time slot is a lost episode. Figure 5.13 shows the overall cumulative distribution of packet loss frequency estimated by Badabing in our spatial and temporal experiments (note the x-axis is in log scale). Here the packet loss frequency is defined as $(loss\_time\_slot/total\_time\_slot)$. From this graph, we can see that Badabing reports abnormally high packet loss frequency in the Amazon EC2 data center. In both spatial and temporal experiment results, more than 10% of the Badabing measurements report very high packet loss frequency ($> 10\%$). This packet loss frequency is extremely high since normally packet loss happens very rarely ($< 0.1\%$). To cross validate, we look at the probing packet traces of all the Badabing measurements, the cumulative distributions of Badabing probe loss rate are also plotted in Figure 5.13. The Badabing probe loss rate is defined as $(lost\_probes/total\_probes)$ in Badabing measurements. From the distribution of Badabing probe loss rate, we can see that probe packet loss actually happens very rarely in both spatial and temporal experiments. For 98% of the measurements, the probe loss rates

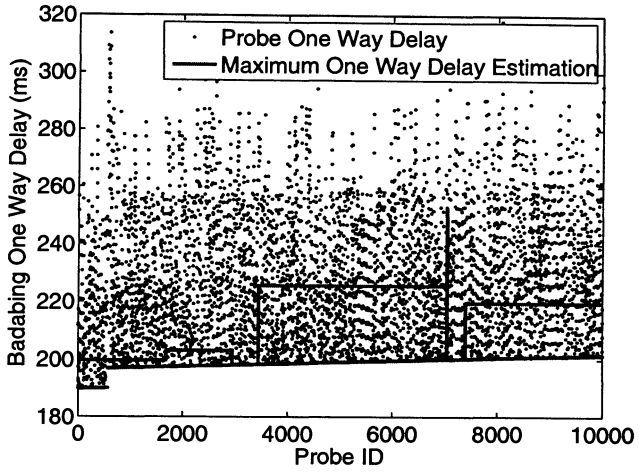


Figure 5.14 : Badabing probe packet one way delay and maximum OWD estimation

are smaller than 0.005 and for 60% of the measurments, the probe loss rates are smaller than 0.001. The very high packet loss frequency reported by Badabing is suspicious. We perform a more detailed discussion on the Badabing estimation results.

Discussion: Badabing estimates the loss characteristics of end-to-end paths by detecting the *loss episodes*. A *loss episode* is defined as the time series indicating a series of consecutive packets (possibly only of length one) were lost [ZDPS01]. There is a sender and a receiver in the Badabing tool. At each 5ms time slot, the sender sends a probe with 30% probability. Each probe includes 3 probing packets and all the probe packets are timestamped. When the receiver receives a probe packet, it simply remembers the packet sequence number and the timestamps. Badabing assumes time synchronization between the sender and receiver. However, since Badabing estimates loss episode based on delay differences, the time synchronization does not have to be perfect. The estimation algorithm marks a time slot as lossy or not lossy based on the one way delay and lost packets in the time slot. The criteria is that if there is a lost packet within τ time of the current time slot, or the one way delay is larger than $max\_owd \times (1-\alpha)$, the time slot is marked as lossy. Here, the max_owd is the maximum one way delay of the path, which is estimated by the one

way delay of the most recent successful packet when a packet loss happens. By default, τ is set to 50 ms and α is set to 0.005. Here, Badabing is making an implicit assumption that when an end-to-end path is in loss episodes, the one way delays (OWD) of this path will be higher than its one way delays when the path is not in loss episodes. This assumption makes sense in the wide area Internet environments.

However, in our previous results, we have observed very high delay variation even though the data center network is not congested. These large delay variations are very likely to be caused by the machine virtualization. The problem is that, the delay variations caused by virtualization can be much larger than the delay variations caused by network congestion. Many of these large delay variations can cause Badabing to mark a time slot as lossy. Therefore, in this environment, Badabing will have a much higher false positive rate. That is why Badabing reports very high packet loss frequency on many instance pairs in our experiments. To demonstrate this effect, we plot the one way delay and corresponding maximum OWD estimation for 10,000 probes on a small instance pair in Figure 5.14. During the 10,000 Badabing probes, there are only 7 packets lost. Every time when a packet loss happens, Badabing will estimate a new maximum OWD based on one way delay of most recent successful packets. From the graph, we can see that the estimated maximum OWDs are not very large. However, in many cases, the one way delay variations caused by virtualization can be much larger than the estimated maximum OWDs. All these cases will cause false positive detections in the Badabing results.

The discussion of Badabing results reveals that, in the virtualized data center environment, there are additional difficulties to infer network properties using statistics. Some valid assumption in traditional network environments may not hold in virtualized data centers.

5.7 Implications

We have found that the networking performance between Amazon EC2 instances demonstrate very different characteristics from traditional non-virtualized clusters, such as the

abnormal large delay variations and unstable TCP/UDP throughput caused by end host virtualization. In this section, we discuss the implications of our findings on the design of virtualization infrastructure and applications running in cloud services.

5.7.1 Implications to cloud applications

Network measurement based systems in cloud: As discussed in the previous section, the large delay variation can completely skew the packet loss estimation results of the Badabing tool. Badabing is just one example of the problem. The fundamental problem is that the simple textbook end-to-end delay model composed of network transmission delay, propagation delay, and router queuing delay is no longer sufficient. Our results show that in the virtualized data center, the delay caused by end host virtualization can be much larger than the other delay factors and cannot be overlooked. Other than the Badabing tool we discussed, the large delay variations can also impact many other protocols and systems that rely on the RTT measurement to infer network congestion, such as TCP vegas [BP95] and PCP [ACKZ06]. Therefore, if the cloud service users want to build systems relying on the network measurements to make decisions, they need to be aware of the virtual machine scheduling characteristics of the virtualized data center environment.

Network experiments in cloud: Emulab is a widely used facility for networking researchers to deploy emulated network experiments. However, Emulab is based on a relatively small computer cluster. In many cases, researchers cannot find enough machines to deploy their large scale experiments. Recently, researchers have proposed to deploy large scale network experiments on the Amazon EC2 service (e.g. the Cloudlab project [Clo]). However, as far as we know, there is no quantitative result about the feasibility of this idea. Our measurement results provide some insights on this problem. To deploy network experiments on Amazon EC2, the challenge is to emulate different kinds of network links between EC2 instances. The processor sharing and unstable network performance bring challenges to the link emulation. First, because all the small EC2 instances are sharing processors with other instances, it is very hard for them to set timers precisely to perform

accurate rate shaping. In addition, the large delay variations and unstable throughput make it hard to emulate stable high speed links among small instances. Using high capacity instances might be able to reduce the problem, but further experimental study is needed to understand this issue.

Scientific computing in cloud: Unstable network throughput and large delay variations can also have negative impact on the performance of scientific computing applications. For example, in many MPI applications, a worker has to exchange intermediate results with all the other workers before it can proceed to the next task. If the network connections to a few workers suffer from low throughput and high delay variations, the worker has to wait for the results from the delayed workers before it can proceed. Therefore, MPI applications will experience significant performance degradation. MapReduce applications [DG04a] may experience the same problem when a large amount of data is shuffled among all the mappers and reducers. To improve the performance of these scientific computing applications on cloud service, we may need to customize their job assignment strategies to accommodate the unstable networking performance among virtual machines.

5.7.2 Improving the virtualization infrastructure

Our results suggest that service providers need to improve the virtualization infrastructure to provide stable and predictable networking performance in cloud. The unstable network performance we observed are mainly caused by the processor sharing and I/O sharing in virtualization infrastructure. There are existing techniques that can be used to address the I/O and processor sharing issues.

To solve the I/O sharing problem, we need to separate the traffic for different guest OSes into different queues and buffers. One technology that can potentially solve the I/O sharing problem is Single Root I/O Virtualization (SR-IOV) [DYR08]. SR-IOV allows a single PCI device to be shared among multiple virtual machines while retaining the performance benefit of assigning a PCI device to a virtual machine. A SR-IOV capable NIC with a single physical network port can be shared by multiple virtual machines by assign-

ing a virtual function to each VM. The traffic sent and received for different VMs will be separated into different queues in the NIC, which can solve the I/O sharing problem we have seen in Xen virtualization platform. However, the SR-IOV technology requires special hardware support and kernel software updates. Wide-deployment of this technology could be expensive and time-consuming in large cloud data centers.

The processor sharing is hard to avoid in virtualization environment. This issue is more about the performance and cost trade-off for cloud users. The whole point of virtualization infrastructure is to allow users share the data center resources efficiently and safely. Amazon EC2 allows users to rent more powerful virtual machines with multiple cores assigned. But it will cost more money for users. Therefore, it is user's choice that if they don't care about the computation performance, the low end virtual machine instances will be sharing processors with other users. For applications with high performance requirement, users may have to rent more powerful virtual machines that are assigned with dedicated processors. One the cloud service provider side, they can test different virtualization platforms, such as Xen, KVM and VMWare, and use the one that with smaller VM scheduling delays.

5.8 Summary

We presented a quantitative study of the end-to-end networking performance among Amazon EC2 instances from users' perspective. We observe wide spread processor sharing, abnormal delay variations and drastically unstable TCP/UDP throughput among Amazon EC2 instances. Through detailed analysis, we conclude that these unstable network characteristics are caused by virtualization and processor sharing on server hosts. The unstable network performance can degrade the performance of and bring new challenges to many applications. Our study provides early results towards understanding the characteristics of virtualized data centers. As future work, we will study how applications can be customized to achieve good performance over virtualized environments.

Chapter 6

Limitations and Future Work

In this chapter, we discuss some of the limitations of our study and potential directions to explore in future work.

6.1 Limitations

Understanding hybrid network at large scale: In the thesis, we have argued that it is feasible to build a hybrid data center network at large scale. But it is still unclear how well the hybrid network would perform in a very large data center. This question is hard to answer due to a few complicated factors. First, as we have discussed, the scheduling cycle for an optical circuit to send traffic to all destinations will be increased significantly. Second, as applications scale up, the traffic concentration properties might change in a large data center. Third, due to the limited size of physical testbed, it is hard to evaluate such a large network with real system and application settings.

Supporting 40/100Gbps optics In this thesis, the highest speed optical interface we tested is 10Gbps optical transceivers. We have shown that applications and the TCP protocol still perform well to support circuit switching of 10Gbps links. But it is unclear how well end-host applications and TCP protocols will perform with optical interfaces at much higher speed, such as 40Gbps or 100Gbps. The major issues are when optical interfaces send at much higher speed, the system will require more frequent circuit reconfiguration and more buffer space at the end-hosts to keep optical circuit highly utilized. It is also unclear how well TCP will perform to support optical interfaces at 40Gbps speed.

The limitations of EC2 measurement One limitation of our measurement study on commercial EC2 cloud is that we can only perform measurement within guest virtual ma-

chine instances. We cannot control the physical location of our virtual machine instances. We cannot control the detailed configuration of virtualization infrastructure and data center network either. So the specific numbers found in our analytical study might not generalize to cloud environments with different virtualization platforms and settings.

6.2 Future Work

The work in the thesis raises a number of interesting issues that can be explored in future work.

6.2.1 Managing optical circuits in virtualized data centers

In the design of c-Through system, we assume no virtualization layer is deployed on servers. c-Through runs a management software and a kernel patch on servers to collect socket buffer occupancy and control traffic. However, in a virtualized data center, it is intrusive to deploy these software components in the virtual machines of cloud users. To manage optical circuits in a virtualized data center, one potential direction to explore is to design traffic demand estimation and traffic control components in virtual machine monitors. There are two advantages of designing traffic control components in virtual machine monitors. First, this can make the optical manager totally transparent to cloud users. Second, we can still leverage both large memory for buffering and programmability on servers to control optical circuits.

Moving traffic demultiplexing component into virtual machine monitors is relatively easy. We can simply move c-Through's multiqueue traffic scheduler into the kernel of virtual machine monitor to schedule traffic to optical or electrical paths. However, measuring traffic demand in virtual machine monitor is more difficult since we cannot enlarge socket buffers and read buffer occupancies of guest OSes from VM monitors. One potential idea is to collect flow counters for individual flows and use iterative Hedera algorithms as in Helios to estimate traffic demand for different virtual machines. With the programmability on servers, we can even implement more intelligent traffic demand estimation for different

applications. But one problem is that existing Hedera algorithm cannot leverage buffering memory on servers to improve circuit utilization, which is question to explore with future work.

6.2.2 Performance modeling and prediction

Our experiences prototyping hybrid data center network architectures reveal the difficulties of evaluating new data center network designs. Building a physical testbed often suffers from the limited scale of testbed and the complexity of configuring distributed applications to generate realistic workloads. A natural alternative would be to build a simulator to evaluate different data center network designs at large scale using synthetic workloads. However, there are two major difficulties in building a simulation platform to evaluate data center network designs. First, how to generate realistic traffic workload for data center networks? The traffic pattern of data center applications are tightly coupled with data center network architecture. Therefore, to generate realistic data center traffic workload, it is important to come up with a traffic model for data center applications, and be able to generate synthetic traffic that can adapt to different data center network designs. Second, how to build a network simulator for large scale data center network? A large scale data center network could have hundreds of thousands of nodes, and various distributed applications. Existing network simulators such as NS-2 cannot support network simulation at this scale. Therefore, building a network simulator for large scale data centers requires designing a good abstract to capture the protocol and flow level behaviors of a large network.

6.2.3 Partial aggregation over optical circuits

As we have discussed in Section 3.8, in a large scale data centers, the hybrid network architecture may suffer from the long optical visit delay in an N-rack data center, we have to wait N circuit reconfiguration rounds to finish all-to-all traffic shuffling in the worst case. This problem could become serious when we run applications with all-to-all data shuffling (e.g. MapReduce and parallel database) in a large data center. One interesting question

to explore in future work would be how we can reduce the worst case optical visit delay to support all-to-all data shuffling more efficiently over limited number of optical circuits. One particular idea we can explore in future work is to use iterative partial aggregation over optical circuits to finish all-to-all data shuffling in $log(N)$ rounds in a N round data center. We explain some initial thoughts as follows.

User-defined partial aggregation: Distributed aggregation is a core primitive for many distributed programming models, such as MapReduce and parallel database. For example, in MapReduce system, the computation has two phrases, Map and Reduce. The user-defined reducer() function collects and merges data from all the mappers. We can view this as full aggregation operation.

However, in distributed programming models, many optimizations can be performed by computing and combining partial aggregations. In [YGI09], Yu et al. discuss the use of partial aggregation in MapReduce execution. The idea is intermediate results can be aggregated among part of the workers first, then merge with another part of the data to get final results. To support partial aggregation in MapReduce, we can consider separating the Reduce function into three phrases:

(1) InitialReduce(): it takes the output of local mappers and does some initial processing.

(2) Combine(): this function takes a sequence of partial aggregations from the InitialReduce output and process the data, generating another set of intermediate results.

(3) FinalReduce(): The FinalReduce function takes the output of Combine or InitialReduce or a mix of them, and performs the final aggregation.

Users can implement different partial aggregations by defining their own combine functions. Yu et al. propose partial aggregation as a flexible computation model in distributed computation applications, showing that this technique can also reduce the network traffic and improve application performance.

Iterative partial aggregation over reconfiguring optical circuits: The idea is to perform partial aggregation over optical circuit connected racks. When optical circuits are

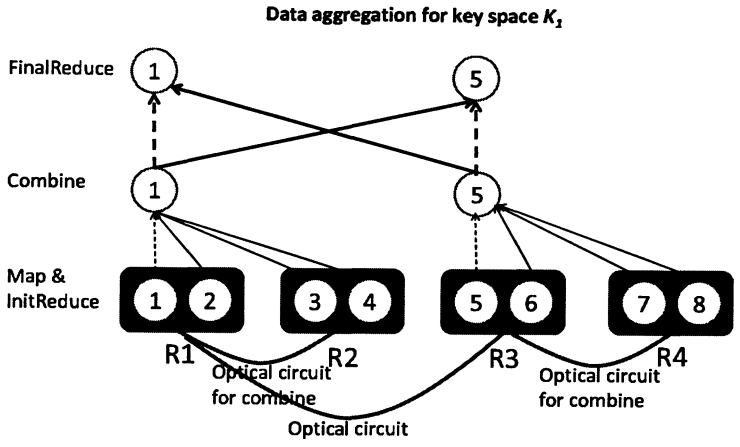


Figure 6.1 : A partial aggregation tree for 4 racks

reconfigured, we do the partial aggregation iteratively until getting the final results. Let's consider an example. As shown in Figure 6.1, suppose we have 4 racks in the data centers, with 2 servers in each rack. The reduce space is divided into 4 partitions (K1, K2, K3, K4). Each of these partitions is assigned to 2 machines, (K1: 1, 5), (K2: 2, 6), (K3: 3, 7), (K4: 4, 8). We can build the aggregation tree in Figure 6.1 to process the data in key space K1.

The execution has three phases. First, all the nodes can perform a local map and InitReduce; second, we set up optical circuits between rack R1 and R2, rack R3 and R4, and perform partial aggregation (Combine) over the optical connected racks. The partial results will be aggregated on server 1 and 5. Third, we set up optical circuits between rack R1 and R3, and do FinalReduce on server 1.

In this example, since there are 4 racks, we need to do only one iteration of partial aggregation. In a larger data center, we can do multiple iterations of partial aggregation. In general, using this partial aggregation technique, we can collect and process all the data in $log(N)$ iterations, where N is the number of racks executing the job. The reduce key space is organized into a hierarchical structure. We need to split the reduce key space into $2S$ partitions, where S is the number of servers in each rack. Each partition is assigned to R

servers, R is the total number of racks, with one server in each rack.

Compared to naively configured optical circuits to support all to all shuffling, the advantage of this design is:

(1) Fewer circuit configurations. This design only needs to reconfigure circuits between different iterations. We only need to reconfigure circuits $log(N)$ times, instead of reconfiguring N times in naive solutions.

(2) More long lasting flows after aggregation. By performing partial aggregation, the high layers of the aggregation tree get more long lasting flows, which is very suitable for optical transfer.

(3) Reducing network traffic for many applications. Of course, we can have the benefit of reducing network traffic by partial aggregation for applications like wordcount and page rank.

6.2.4 Performance monitoring and diagnosis in virtualized cloud

Our EC2 measurement results reveals that processor sharing and I/O sharing in virtualization layer cause unstable bandwidth and delay variations from end-to-end user's point of view. Our findings inspire a set of interesting questions about performance monitoring and diagnosis in virtualized cloud that can be explored in future work. The first question is how can we diagnose cloud data center network by end-to-end measurement from virtual machine instances. Since many cloud users may run distributed applications in the cloud, users may need to diagnose the network performance in virtualized clouds in many scenarios, such as debugging distributed applications. However, as we have discussed in previous sections, it becomes difficult to infer network properties from end-to-end measurement in virtual machines due to the unstable network performance caused by virtualization layer.

Another question is how can we monitor the traffic of cloud users and detect any failures and performance issues that might be experienced by users. To provide cloud service with high availability, it is important that we can detect and locate any failure and performance problems happened in cloud data centers. This problem is hard to solve because of the

large number of nodes, application components and system layers involved in the system. We may have to build a monitoring infrastructure that can collect traffic information from both switches and virtual machine monitors, and perform timing-based analysis to detect performance issues. Many detailed system and algorithm design issues need to addressed for this problem.

Chapter 7

Conclusion

The rising tide of cloud service and data intensive applications brings significant challenges to the design of data center network infrastructure on both physical interconnect layer and virtualization layer. This dissertation sought to understand the design and performance implications of data center network infrastructure for cloud services. We explore the use of optical switches to provide high bandwidth in data centers and the implications of virtualization layers on network performance of cloud data centers. We find that optics has great promises in constructing high bandwidth data center network at low cost and low complexity. Our analytical study also reveals that a virtualization layer has significant implications on the network performance experienced by cloud users. The findings of the thesis provide important insights in the design of data center network infrastructure for high performance cloud services. More specifically, the thesis makes the following contributions.

The design of HyPaC network and its applicability: We proposed a hybrid packet/circuit switched data center network architecture which leverages high capacity optical switches to provide high bandwidth for data intensive applications. We presented the basic system design and show that it is feasible to construct a hybrid data center network without modifying today's Ethernet switches and data center applications. We built a prototype system called c-Through and perform an empirical study to understand the applicability of the hybrid data center network. We showed that the hybrid network can potential provide close-to-optimal performance even for applications with all-to-all communication patterns like MapReduce.

Circuit control in shared data centers: A further effort in the thesis tries to address the obstacles in adopting optical circuits in real data center environment. We identified the

challenges of using optical circuits in heterogeneous data centers. We presented a flexible control framework to manage optical circuits for non-cooperative applications sharing the data center. We discussed the algorithms to schedule optical circuit among application with interleaving traffic dependencies.

Network performance in virtualized cloud: We presented an analytical study to understand the implications of virtualization on network performance in Amazon EC2 data centers. Our results reveal significantly unstable bandwidth and delay variations caused by the virtualization layer in data centers. Our findings provide valuable insights for both cloud users and service providers on how to adapt applications in cloud environments and how to improve the virtualization infrastructure for cloud services.

The results in the thesis point to new directions and raise many questions about the design of data center network infrastructure for cloud services. With the explosion of bandwidth demand in and across cloud data centers, we believe optics will play an important role in supporting high bandwidth communication in cloud data centers. Our study of hybrid data center network open a research rich topic of how to leverage optical switching technology to build high performance, power efficient data center network at low cost. There are many potential problems to explore on both application design, network control and physical interconnect layers. While virtualization is becoming a key infrastructure in cloud, our results on the network performance of virtualized data centers raise many research questions on the design of virtualization layer. Many of these problems, such as performance isolation and application adoption in virtualized environment, will be major challenges in the wide-spread deployment of cloud services.

Bibliography

[ACKZ06] T. Anderson, A. Collins, A. Krishnamurthy, and J. Zahorjan. Pcp: Efficient endpoint congestion control. In *Proceedings of USENIX NSDI'06*, May 2006.

[AFG+09] Michael Armbrust, Armando Fox, Rean Grifth, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical report, University of California at Berkeley, February 2009.

[AFLV08] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of SIGCOMM'08*, August 2008.

[AFRR+10] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, April 2010.

[AK89] E. Aarts and J. Korst. Simulated annealing and boltzmann machines. In *J. Wiley Sons*, 1989.

[Ama] Amazon ec2. http://aws.amazon.com/ec2/.

[And09] Dragos Andrei. Efficient provisioning of data-intensive applications over optical networks. Fall 2009. Ph.D. thesis, UC. Davis.

[BAAZ09a] T. A. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. In *Proceedings of SIGCOMM WREN Workshop*, August 2009.

[BAAZ09b] T. A. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. In *Proc. Workshop: Research on Enterprise Networking*, Barcelona, Spain, August 2009.

[BBea05] K. Barker, A. Benner, and R. Hoare et al. On the feasibility of optical circuit switching for high performance computing systems. In *Proc. SC05*, 2005.

[BDF+03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of SOSP'03*, October 2003.

[BK73] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. In *Communications of the ACM*, September 1973.

[BM06] Jens Buus and Edmond J. Murphy. Tunable lasers in optical networks. *Journal of Lightwave Technology*, 24(1), 2006.

[BP95] L. S. Brakmo and L. Peterson. Tcp vegas: End-to-end congestion avoidance on a global internet. *IEEE Journal of Selected Areas in Communication*, 13(8), October 1995.

[BR01] Anindya Basu and Jon G. Riecke. Stability Issues in OSPF Routing. In *Proc. ACM SIGCOMM*, San Diego, CA, August 2001.

[CGA+06] Martin Casado, Tal Garfinkel, Aditya Akella, Michael J. Freedmani, Dan Boneh, Nick McKeown, and Scott Shenker. SANE: A Protection Architecture for Enterprise Networks. In *Proceedings of the 2006 Usenix Security Symposium*, 2006.

[Clo] Cloudlab. http://www.cs.cornell.edu/ einar/cloudlab/index.html.

[CR99] W. Cook and A. Rohe. Computing minimum-weight perfect matchings. *INFORMS Journal of Computing*, 11:138–148, 1999.

[DC99] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time(bvt) scheduling: supporting latency-sensitive threads in a general purpose scheduler. In *Proceedings of SOSP'99*, December 1999.

[DG04a] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of USENIX OSDI'04*, December 2004.

[DG04b] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. 6th USENIX OSDI*, San Francisco, CA, December 2004.

[DHJ+07] Guiseppe DeCandia, Deinz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, October 2007.

[DPS+10] Saurav Das, Guru Parulkar, Preeti Singh, Daniel Getachew, Lyndon Ong, and Nick Mckeown. Packet and circuit network convergence with openflow. In *Optical Fiber Conference (OFC/NFOEC'10)*, March 2010.

[DYR08] Yaozu Dong, Zhao Yu, and Greg Rose. SR-IOV Networking in Xen: Architecture, Design and Implementation. In *First Workshop on I/O Virtualization*, San Diego, CA, December 2008.

[ECN06] Khaled Elmeleegy, Alan Cox, and T.S. Eugene Ng. On Count-to-inifity Induced Forwarding Loops in Ethernet Network. In *Proc. IEEE INFOCOM*, Barcelona, Spain, March 2006.

[Edm65] J. Edmonds. Paths, trees and flowers. *Canadian Journal on Mathematics*, pages 449–467, 1965.

[EYD07] D. Ersoz, M. S. Yousif, and C. R. Das. Characterizing network traffic in a cluster-based, multi-tier data center. In *Proceedings of ICDCS'07*, June 2007.

[FFL+11] Nathan Farrington, Yeshaiahu Fainman, Hong Liu, George Papen, and Amin Vahdat. Hardware requirement for optical circuit switched data center networks. In *Optical Fiber Conference (OFC/NFOEC'11)*, March 2011.

[FP01] C. A. Floudas and P. M. Pardalos. Heuristics for maximum clique and independent set. In *Encyclopedia of Optimization, Kluwer Academic Publishers, Boston, MA, Vol. 2, pp.411-423,*, 2001.

[FPR+10] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *ACM SIGCOMM*, August 2010.

[GAKM09] Madeleine Glick, David G. Andersen, Michael Kaminsky, and Lily Mummert. Dynamically reconfigurable optical links for high-bandwidth data center networks. In *Optical Fiber Comm. Conference (OFC)*, March 2009.

[Gal86] Zvi Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Computing Survey*, 18:23–38, 1986.

[Gar07] Simson L. Garfinkel. An evaluation of amazon's grid computing services: Ec2, s3 and sqs. Technical report, Computer Science Group, Harvard University, 2007. TR-08-07.

[GHM+05] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin. Zhan, and Hui Zhang. Clean Slate 4D Approach to Network Control and Management. In *Proceeding of ACM SIGCOMM Computer Communication Review*. ACM, 2005.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W.H. Freeman and Company, 1979.

[GJK+09] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *Proc. ACM SIGCOMM* [sig09].

[GKP+08] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martn Casado, Nick McKeown, and Scott Shenker. NOX: towards an operating system for networks. In *Proceedings of ACM SIGCOMM '08 Computer Communication Review*. ACM, July 2008.

[GLL+09] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A high performance, server-centric network architecture for modular data centers. In *Proc. ACM SIGCOMM* [sig09].

[GLW+10] Chuanxiong Guo, Guohan Lu, Helen Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. Secondnet: A data center network virtualization architecture with bandwidth guarantees. In *ACM CoNEXT'10*, December 2010.

[GoG] Gogrid. http://www.gogrid.com/.

[GWT+08a] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: A scalable and fault-tolerant network structure for data centers. In *Proceedings of SIGCOMM'08*, August 2008.

[GWT+08b] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. DCell: A scalable and fault-tolerant network structure for data centers. In *Proc. ACM SIGCOMM* [sig08].

[had] Apache Hadoop, http://hadoop.apache.org.

[HALOD10] Antony Rowstron Hussam Abu-Libdeh, Paolo Costa, Greg O'Shea, and Austin Donnelly. Symbiotic routing in future data centers. In *Proceedings of SIGCOMM'10*, August 2010.

[hot09] New York City, NY. USA., October 2009.

[HP96] S. Homer and M. Peinado. Experiments with polynomial-time clique approximation algorithms on very large graphs. In *DIMACS Series. American Mathematical Society*, 1996.

[JCH84] R. Jain, D.M. Chiu, and W.R. Hawe. *A quantitative measure of fairness and discrimination for resource allocation in shared computer system.* Eastern Research Laboratory, Digital Equipment Corp., 1984.

[Jos09] Joseph Berthold. Optical networking for data center interconnects across wide area networks, 2009. `http://www.hoti.org/hoti17/program/slides/SpecialSession/HotI-2009-Bert%hold.pdf`.

[KCR08] Changoon Kim, Matthew Caesar, and Jennifer Rexford. Floodless in seattle: A scalable ethernet architecture for large enterprises. In *Proc. ACM SIGCOMM* [sig08].

[KNK+03] J. Kim, C.J. Nuzman, B. Kumar, D.F. Lieuwen, et al. 1100x1100 port MEMS-based optical crossconnect with 4-dB maximum loss. *IEEE Photonics Technology Letters*, pages 1537–1539, 2003.

[KPB09] Srikanth Kandula, Jitendra Padhye, and Victor Bahl. Flyways to de-congest data center networks. In *Proc. ACM Hotnets-VIII* [hot09].

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR*, 2008.

[Mck99] Nick Mckeown. The iSLIP scheduling algorithm for input-queued switches. *IEEE/ACM Transactions on Networking*, 7(2):188–201, April 1999.

[MCZ06] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in xen. In *Proceedings of USENIX'06*, June 2006.

[MFP+07] Casado Martin, M J Freedman, Justin Pettit, J Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *Proceedings of ACM SIGCOMM '07 Computer Communication Review*. ACM, 2007.

[MNZ04] Andy Myers, T.S. Eugene Ng, and Hui Zhang. Rethining the service model: Scaling ethernet to a million nodes. In *Proc. 3nd ACM Workshop on Hot Topics in Networks (Hotnets-III)*, San Diego, CA, November 2004.

[MPF+09] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: A scalable fault-tolerant layer2 data center network fabric. In *Proc. ACM SIGCOMM* [sig09].

[MST+05] A. Menon, J. R. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel. Disgnosing performance overheads in the xen virtual machine environment. In *Proceedings of VEE'05*, June 2005.

[OCR08] Diego Ongaro, Alan L. Cox, and Scott Rixner. Scheduling i/o in virtual machine monitors. In *Proceedings of VEE'08*, March 2008.

[QY99] C. Qiao and M. Yoo. Optical Burst Switching (OBS) - A New Paradigm for An Optical Internet. *Journal of High Speed Networks*, 8(1):69–84, 1999.

[RHHR09] Costin Raiciu, Felipe Huici, Mark Handley, and David S. Rosen. Roar: Increasing the flexibility and performance of distributed search. In *Proceedings of SIGCOMM'09*, August 2009.

[RPC+11] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat.

Tritonsort: A balanced large-scale sorting system. In *USENIX NSDI'11*, March 2011.

[SBDR05] J. Sommers, P. Barford, N. Duffield, and A. Ron. Improving accuracy in end-to-end packet loss measurement. In *Proceedings of SIGCOMM'05*, August 2005.

[SGNC04] S. Sharma, K. Gopalan, S. Nanda, and T. Chiueh. Viking: A multi-spanning-tree ethernet architecture for metropolitan area and cluster networks. In *Proc. IEEE INFOCOM*, Hong Kong, March 2004.

[sig08] Seattle, WA, August 2008.

[sig09] Barcelona, Spain, August 2009.

[SKGK11] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Sharing the data center network. In *USENIX NSDI'11*, March 2011.

[Sys08] Cisco Systems. Data Center: Load Balancing Data Center Service. In *Cisco Documents*, November 2008.

[SZW+09] L. Schares, X.J. Zhang, R. Wagle, D. Rajan, P. Selo, S. P. Chang, J. Giles, K. Hildrum adn D. Kuchta, J. Wolf, and E. Schenfeld. A reconfigurable interconnect fabric with optical cicuit switch and software optimizer for stream computing systems. In *Optical Fiber Comm. Conference (OFC)*, March 2009.

[Tur99] J. Turner. Terabit Burst Switching. *Journal of High Speed Networks*, 8(1):3–16, 1999.

[VPS+09] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. In *Proceedings of SIGCOMM'09*, August 2009.

[WAK+09] G. Wang, D. Andersen, M. Kaminsky, M. Kozuch, T. S. E. Ng, K. Papagiannaki, M. Glick, and L. Mummer. Your datacenter is a router: The case for reconfigurable optical circuit switched paths. In *Proc. ACM Hotnets-VIII* [hot09].

[Wal08] Edward Walker. Benchmarking amazon ec2 for high-performance scientific computing. In *USENIX ;login: magzine*, October 2008.

[WBR11] Richard Wang, Dana Butnariu, and Jennifer Rexford. Openflow-based server load balancing gone wild. In *Workshop of HotICE'11*, March 2011.

[wwwa] Alcatel-Lucent Bell labs announces new optical transmission record and breaks 100 Petabit per second kilometer barrier. `http://tinyurl.com/yau3epd`.

[wwwb] Calient networks diamondwave fiberconnect. `http://www.calient.com/products/diamondwave_fiberconnect.php`.

[wwwc] FFTW: Fastest Fourier Transform in the West. `http://www.fftw.org/`.

[wwwd] Google cluster data. `http://code.google.com/p/googleclusterdata/`.

[wwwe] Gridmix: Trace-based benchmark for mapreduce. `https://issues.apache.org/jira/browse/MAPREDUCE-776`.

[wwwf] IETF transparent interconnection of lots of links (TRILL) working group. `http://datatracker.ietf.org/wg/trill/charter/`.

[wwwg] KVM: Kernel Based Virtual Machine. `http://www.linux-kvm.org/page/Main_Page`.

[wwwh] OpVista CX8 Optical Networking System for 40G and 100G Services to debut at NXTComm08. `http://www.encyclopedia.com/doc/1G1-180302517.html`.

[wwwi] The OpenCirrus Testbed. `https://opencirrus.org/`.

[wwwj] UCLP Project. `http://www.uclp.ca/`.

[www03] Glimmerglass 80x80 MEMS switch. Website, 2003. http://electronicdesign.com/article/test-and-measurement/3d-mems-based-optical-switch-handles-80-by-80-fibe.aspx.

[YGI09] Yuan Yu, Pradeep K. Gunda, and Michael Isard. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[YQD00a] M. Yoo, C. Qiao, and S. Dixit. QoS Performance of Optical Burst Switching in IP-over-WDM Networks. *Journal on Selected Area in Communications*, 18:2062–2071, 2000.

[YQD00b] Myungsik Yoo, Chunming Qiao, and Sudhir Dixit. QoS Performance of Optical Burst Switching in IP-Over-WDM Networks. *IEEE Journal of Selected Areas in Communications, Vol.18, No.10*, 2000.

[YQD01] Myungsik Yoo, Chunming Qiao, and Sudhir Dixit. Optical Burst Switching for Service Differentiation in the Next-Generation Optical Internet. *IEEE Communication Magzine*, February 2001.

[ZDPS01] Y. Zhang, N. Duffield, V. Paxson, and S. Shenker. On the constancy of internet path properties. In *Proceedings of IMW'01*, November 2001.

[ZWG09] Xiaolan Zhang, Rohit Wagle, and James Giles. VLAN-based routing infrastructure for an all-optical circuit switched lan. In *IBM T.J. Watson Research Center Research Report*, July 2009.